TRABAJO DE FIN DE GRADO

Análisis visual de la calidad del aire de Madrid

Autor: Daniel Fernández

Tutora: Teresa Onorati

GRADO EN INGENIERÍA INFORMÁTICA

25 de septiembre de 2018

Resumen

En este trabajo se presenta el diseño, implementación y pruebas de un conjunto de visualizaciones interactivas de la calidad del aire en la ciudad de Madrid. Este conjunto de visualizaciones se presenta al usuario en forma de *dashboard*[1] accesible a través de una página web, donde se puede consultar el estado de la contaminación del aire de Madrid por zonas en un intervalo de tiempo específico, así como otras variables que pueden influir de manera más o menos directa en este índice.

Además del índice de calidad del aire, otros datos que se pueden consultar en la web desarrollada son

1. La temperatura en distintos puntos de medición
2. La presión atmosférica
3. La intensidad media del tráfico en las calles cercanas al punto de medición
4. Diferentes valores para la cantidad de partículas presentes en la atmósfera que, a través de una fórmula, constituyen el índice de calidad del aire

Este trabajo se ha llevado a cabo en cinco fases distintas: investigación, análisis, diseño y desarrollo. La versión original del trabajo parte de un proyecto personal, pero se ha iterado sobre unos requisitos obtenidos y una serie de características propuestas sobre las que se basa el diseño actual hasta llegar hasta el resultado final.

El *stack*[2] tecnológico utilizado para el desarrollo de este producto está basado en tecnologías web (*frameworks frontend*) como Reactjs para la parte visual o del cliente, acompañado de varias librerías de visualización y tratamiento de datos como Turf o D3.js; Node.js para la parte *back* del proyecto (es decir, el componente de servidor) y mongoDB (base de datos noSQL) para la base de datos que se utiliza tanto en la recolección de datos como en su análisis y representación.

[1] panel informativo

[2] conjunto de tecnologías escogidas en la resolución de un proyecto

Agradecimientos

Quiero dar las gracias ante todo a mi familia. A Antonio y Juana, mis padres, por haber hecho todo lo posible para que hoy esté donde estoy. También por los "deberes", por las lecturas de esta memoria, y por ayudarme en todo momento con cualquier cosa que se tercie. A Cris, mi pareja, por motivarme a ser una versión mejor de mí mismo cada día, y por apoyarme en todo lo que me ha hecho falta, siempre.

Quiero dar las gracias también a mis amigos, de la universidad y fuera de ella, por contar conmigo y por animarme -directa o indirectamente- con la realización de este proyecto. A las personas que han colaborado indirectamente con este trabajo (con opiniones, en las pruebas de usuario...), también, mil gracias.

También quiero dar las gracias a mis profesores, a aquellos que le ponen interés en transmitir lo que saben, que les apasiona su asignatura y saben transmitirlo a los alumnos. A vosotros, gracias por enseñarme y por hacer de ese esfuerzo algo más llevadero.

En especial, quiero dar las gracias a mi tutora, Teresa, que me apoyó desde el primer momento con la idea que le llevé, y me ayudó con el material y a darle forma al trabajo.

Índice

Índice de figuras

Índice de tablas

1. Introducción

En esta sección se describen las bases del trabajo que se presenta, tales como los objetivos de este proyecto, la estructura del presente documento, el impacto de este trabajo y su marco legal.

1.1. Motivación

El modelo de asentamiento de la población [1] preferentemente en zonas urbanas antes que en zonas rurales, ha propiciado que la población se concentre en mayor número en grandes urbes como Madrid y su área metropolitana.

Para Sergio Bailén [2], "La mayoría de la literatura relacionada y de casos de estudio, está de acuerdo con que el modelo de ciudad concentrada es la más sostenible, en contraposición con uno donde la ciudad crece de manera dispersa, porque consume menos suelo, no se extienden las redes de servicios públicos, ya movilidad es más eficiente, entre otras razones [...] El ecourbanismo, o urbanismo basado en consideraciones ambientales y sociales, ha tenido gran acogida en esta última década, debido a su importancia para la sostenibilidad urbana".

Tal concentración de población, con la consecuente elevación del número de vehículos que fluyen y, en general, el consumo derivado de este nivel de vida, suponen un incremento [3] de ciertas partículas contaminantes en la atmósfera, que terminan por provocar problemas de salud a los habitantes.

Para Salvador Rueda[4], "el urbanismo ecológico se diferencia del ortodoxo en cuanto a que posee un sistema de indicadores y condicionantes (restricciones) que se aplican en la eficiencia del sistema o en la habitabilidad y que están determinados por el contexto y sus escalas de intervención [...] La habitabilidad urbana está relacionada con la mejora de la calidad de vida y confort de todos los seres vivos en un contexto temporal, cultural y geográfico".

A menudo se pretende paliar el alto nivel de la contaminación atmosférica en las ciudades con planes de gestión política municipal, que implican "medidas estructurales y tecnológicas concretas que resulten en una significativa reducción de emisiones" [5].

Un ejemplo de estas medidas es la restricción al tráfico en los accesos a la zona centro de Madrid en días de altos niveles de contaminación atmosférica. Éste es uno de los motivos por los que, en determinadas ocasiones, se necesita hacer una rápida consulta del estado de la contaminación atmosférica de la zona donde vivimos o trabajamos.

Además de esto, es evidente el impacto de estos factores en la salud de los habitantes de Madrid[6], por lo que la posibilidad de monitorizar estos factores para conocer la situación de la calidad del aire es otro motivo a tener en cuenta de cara a evitar zonas con altos niveles de contaminación.

1.2. Objetivos generales

Este trabajo pretende facilitar la consulta de los datos sobre la contaminación atmosférica en diferentes zonas de Madrid, y ayudar a entender las correlaciones entre diferentes variables que pueden afectar al aumento de dicho nivel de contaminación y su evolución a lo largo del tiempo.

El proyecto se diseña de manera que pueda satisfacer a distintos tipos de usuarios. Por un lado, el caso de un usuario común que pueda acceder a la aplicación para encontrar información relativa al estado de la calidad del aire en una zona de interés, y quizás explorar los datos de forma rápida para satisfacer su curiosidad. Por otro lado, se dirige hacia un tipo de usuario experto o avanzado, que usa la herramienta con fines exploratorios, con el objetivo de analizar los factores que más influyen en el índice de calidad del aire, y su evolución a lo largo de diversas series temporales.

Por ello, el diseño e implementación del proyecto debe resultar sencillo y claro de usar, manteniendo una complejidad opcional para el usuario avanzado que la necesite. Asimismo este proyecto deberá ser accesible, con versiones de la web adaptadas a diversas plataformas (móvil, tablet o escritorio).

Además de los citados objetivos generales, los objetivos específicos que se fijan para este proyecto son los siguientes:

1. Mostrar y tratar la mayor cantidad de información disponible en diversos grados de complejidad, usando técnicas de visualización de datos para facilitar el análisis visual.

2. Seguir un proceso de diseño basado en la investigación, con el objetivo de definir el proyecto a realizar y su proceso de desarrollo.

3. Seguir un proceso de desarrollo basado en la filosofía *open source*, siguiendo estándares de calidad y formato para el código, que estará disponible de forma pública.

4. La integración y documentación de todas las capas del proyecto, desde la capa *back* del servidor y su base de datos, hasta la capa *front* visual del *dashboard* y su integración con el anterior.

1.3. Estructura del documento

Este documento se divide de la siguiente forma:

1. Introducción: como se ha explicado antes, se exponen la motivación, los objetivos y las metas de este proyecto, la estructura del documento que lo acompaña y posible impacto del proyecto.

2. Estado del arte: en esta sección se presentan aspectos contextuales del proyecto y se detallan las tecnologías usadas en la implementación de este proyecto, así como diversas aplicaciones disponibles y aproximaciones similares al proyecto desarrollado.

3. Arquitectura del sistema y diseño de la solución: esta sección propone una aproximación a la solución del problema descrito, así como un detalle de los requisitos y del entorno de desarrollo. Además, detalla el proceso de implementación del proyecto final.

4. Evaluación: esta sección valora el proceso de evaluación y test del proyecto realizado, las pruebas realizadas y su entorno.

5. Gestión del trabajo: esta sección trata sobre la gestión de recursos humanos y materiales del proyecto, en términos de costes de tiempo y dinero.

6. Consideraciones y conclusiones: este capítulo cierra el presente documento con las conclusiones del proyecto desarrollado, así como consideraciones a tener en cuenta a lo largo de todas las fases de vida del proyecto, y conclusiones personales sobre el mismo.

7. Por último, el documento incluye un resumen detallado de la misma memoria del trabajo, traducido al inglés.

1.4. Marco regulador

Esta subsección del documento hace referencia a cualquier normativa que pueda afectar directa o indirectamente al trabajo presentado, o tener relación con este. En este caso, existe una normativa legal que afecta directa o indirectamente al trabajo presentado, tanto sobre el uso de datos de las APIs utilizadas en el proyecto, como al estado de la calidad del aire. Concretamente, en el caso de Madrid, también existe un Protocolo de actuación en episodios de alta contaminación atmosférica.

1.4.1. Normativa de uso de las APIs

El marco legal que regula el uso y condiciones de las APIs empleadas en este proyecto (como son la API de la Agencia Estatal de Meteorología -AEMET-, y la API del Portal de datos abiertos del Ayuntamiento de Madrid) es el siguiente:

1. Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal
2. Real Decreto 1720/2007, de 21 de diciembre, por el que se aprueba el Reglamento de desarrollo de la Ley Orgánica 15/1999, de 13 de diciembre, de protección de datos de carácter personal.
3. Reglamento (UE) 2016/679 del Parlamento Europeo y del Consejo, de 27 de abril de 2016, relativo a la protección de las personas físicas en lo que respecta al tratamiento de datos personales y a la libre circulación de estos datos y por el que se deroga la Directiva 95/46/CE (Reglamento general de protección de datos).
4. Real Decreto-ley 5/2018, de 27 de julio, de medidas urgentes para la adaptación del Derecho español a la normativa de la Unión Europea en materia de protección de datos.

En general, tanto AEMET como el Ayuntamiento de Madrid delegan la responsabilidad del uso de los datos en el usuario. En los términos y condiciones de acceso a los datos de AEMET se puede leer: “El acceso a este portal web, así como el uso de la información que contiene, son de la exclusiva responsabilidad del usuario.”[7].

En cuanto al propio uso de los datos, en los términos y condiciones de AEMET, encontramos que "El usuario se compromete a utilizar esta página o las diferentes formas de acceso, sin incurrir en actividades que puedan ser consideradas ilícitas o ilegales, que infrinjan los derechos de AEMET o de terceros, o que puedan dañar, inutilizar, sobrecargar o deteriorar el sitio web u otras formas de acceso o impedir la normal utilización del mismo".

Igualmente, en la página homóloga del Ayuntamiento de Madrid[8] podemos encontrar: "La utilización de los conjuntos de datos se realizará por parte de las personas y/o empresas que reutilizasen datos, bajo su propia cuenta y riesgo, correspondiéndoles en exclusiva a ellos responder frente a terceros por daños que pudieran derivarse de ella.".

1.4.2. Normativa sobre Calidad del Aire y Protocolo del Ayuntamiento de Madrid

1. Ley 34/2007, de 15 de noviembre, de Calidad del Aire y Protección de la Atmósfera, que contempla la obligación de "informar a la población sobre los niveles de contaminación y calidad del aire"

2. Real Decreto 102/2011, de 28 de enero, relativo a la Mejora de la Calidad del Aire, que establece "umbrales de alerta para tres contaminantes" y define dicho umbral como "el nivel a partir del cual una exposición de breve duración supone un riesgo para la salud humana".

3. Protocolo de medidas a adoptar durante episodios de alta contaminación por dióxido de nitrógeno[9]. Dicho protocolo es una norma elaborada por el Ayuntamiento de Madrid, para la información, definición de "escenarios posibles", medidas informativas y actuaciones a realizar en cada caso.

1.5. Entorno socio-económico

Este trabajo no pretende incidir sobre la actividad económica o buscar un beneficio económico particular en un área concreta – más allá de las consecuencias que de los análisis se pudieran extraer indirectamente – sino que, más bien, afecte a otros tipo de valores sociales y comunitarios, como la toma de conciencia sobre la relación entre contaminación, salud y calidad de vida; o sobre cómo puede contribuir la actividad y los hábitos del individuo en el estado y evolución del aire, en entornos urbanos de alta densidad de población. Todo ello en su doble vertiente de proporcionar una información rápida y directa, y de inducir al usuario a profundizar en las causas de los problemas de la contaminación atmosférica.

Por esta razón, con este proyecto presentamos una solución que cumple una doble función: de comunicación y divulgación sobre las distintas causas de contaminación del aire en Madrid, y de exploración y análisis más en detalle acerca de las partículas que componen el análisis de esta calidad del aire. Esto pretende cubrir dos perfiles: el del usuario medio, que busca informarse sobre el estado de la calidad del aire y aprender sobre tendencias y relaciones entre las otras variables y el potencial impacto en este índice; y el del usuario avanzado, que puede hacer uso de esta herramienta para apoyar el análisis sobre estos factores y su impacto, o para entender más en detalle el impacto de ciertos factores en el aumento o disminución de ciertas partículas que afectan a esta medida.

2. Estado del arte

En este capítulo se detallan las tecnologías y herramientas utilizadas en la realización del proyecto, se analizan las alternativas disponibles hoy en día al trabajo realizado, y se explican algunos conceptos básicos para el desarrollo del proyecto.

2.1. Historia de la cuestión

Para hablar sobre la visualización de datos, es conveniente hacer una pequeña introducción previa acerca del significado e implicaciones del término "dato".

La palabra **dato** (del latín *datum*, que significa "algo dado") se reconoce como **unidad de información**. Dicha unidad informativa puede ser un número, una persona, una característica, etc. [10].

Los datos se pueden dividir a grandes rasgos en tres tipos:

1. Datos nominales: objetos, nombres, conceptos. Las preguntas que aplican sobre estos datos son "qué" y "dónde". Estos datos no tienen una relación cuantitativa implícita o un orden inherente. Los datos nominales también pueden compartir características que permitan agruparlos. Debido a estas categorías, a menudo se les considera también **datos categóricos**. Normalmente estos datos van acompañados por otros tipos de datos que les dan significado u orden.

2. Datos ordinales: estos datos señalan el orden o clasificación, pero no el grado de diferencia entre elementos.

3. Datos cuantitativos: estos datos se pueden medir y, por tanto, se pueden manipular numéricamente (e.g. con métodos estadísticos). Estos datos tienen magnitudes asociadas y responden a la pregunta de "cuánto". Los datos cuantitativos o **datos numéricos** se pueden transformar en datos ordinales categorizándolos (agrupando significativamente).

Representar estructuras de datos multidimensionales de forma visual en dos dimensiones no es algo trivial. El proceso de diseño requiere métodos de razonamiento tanto analíticos como visuales/espaciales. El diseño de información, por lo tanto, depende de los procesos cognitivos y de la percepción visual tanto para la creación (codificación) como para su interpretación (decodificación).

La **visualización de datos** se puede entender a través de dos términos: **infografías** y **diseño de la información**[10].

1. Infografías: son visualizaciones en las que los gráficos comunican una información que no podría comunicarse de otro modo. Ejemplos: mapas del tiempo, diagramas del cuerpo humano, etc.

2. Diseño de información: se utiliza para describir las prácticas de diseño en las que el objetivo principal es informar.

   Las infografías son uno de los resultados del diseño de información. El objetivo final es revelar patrones y relaciones desconocidas que no son fáciles de deducir sin la ayuda de la representación visual de la información.

Actualmente, gracias a la tecnología, disponemos de visualizaciones de datos interactivas y dinámicas, capaces de comunicar mayor cantidad de información de manera más clara al usuario.

Según Card y otros[11], la **visualización de datos** y la **visualización de información** son dos términos que se refieren al "uso de representaciones visuales, informatizadas e interactivas, de datos abstractos para aumentar el conocimiento".

2.2. Trabajos similares: herramientas de análisis de calidad del aire

Tanto a nivel nacional como internacional, existen ya aplicaciones o páginas web similares que abordan la problemática de la información sobre la contaminación atmosférica en distintos territorios. Algunas de ellas son:

- Portal web de Calidad del Aire del Ayuntamiento de Madrid:

  El Departamento del Servicio de Calidad del Aire del Ayuntamiento de Madrid pone a disposición del ciudadano una página web[12] en la que se puede consultar la medición de cada estación de meteorología con un retraso de aproximadamente una hora. Aunque sí dispone de un histórico del Índice de Calidad del Aire y de los valores de las partículas, no indica el estado de este índice en puntos geográficos concretos (es decir, la medición más cercana correspondiente a un punto geográfico), y sólo se pueden consultar datos relativos a las partículas contaminantes. En las figuras 1 y 2 podemos ver la interfaz de esta página web, el aspecto general de la interfaz y el detalle del mapa a pantalla completa.

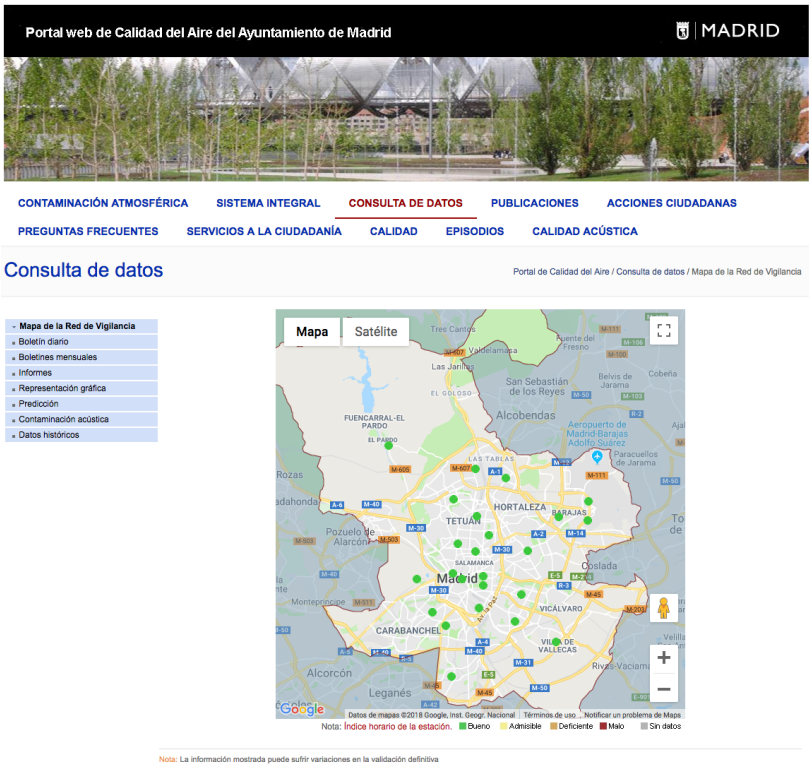

Figura 1: Portal web de Calidad del Aire del Ayuntamiento de Madrid

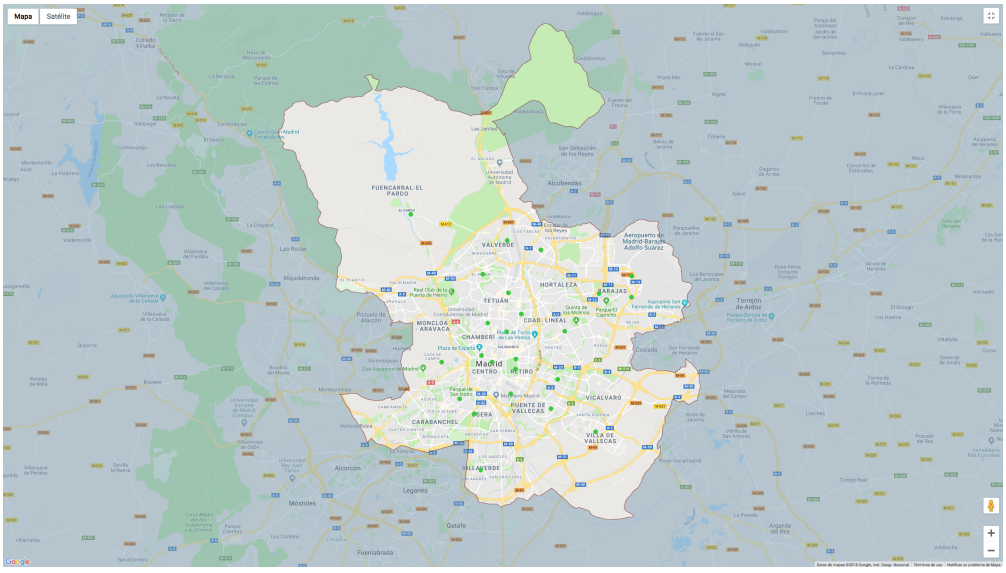

Figura 2: Portal web de Calidad del Aire (Ayto. de Madrid), detalle de la visualización del mapa a pantalla completa

- eltiempo.es:

  Esta página web[13] tiene una visualización parecida a la anterior, con una tabla de partículas contaminantes (con sus magnitudes correspondientes) y un valor del ICA, o Índice de Calidad del Aire, así como información sobre cómo se calcula este índice. En comparación con el proyecto presentado, no da información sobre otros factores relevantes (en la misma página), ni muestra un histórico de los datos. En la figura 3 podemos ver la interfaz de esta página en dos secciones montadas en horizontal.

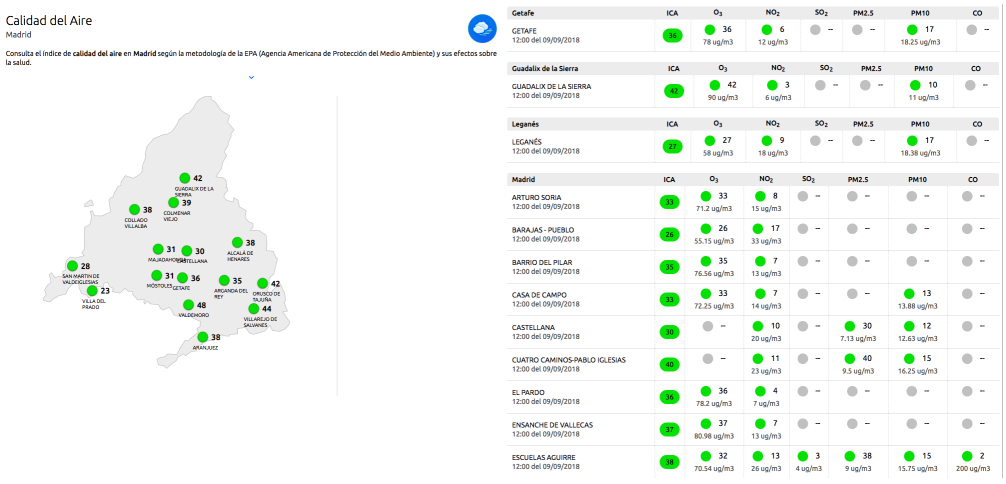

Figura 3: Web de eltiempo.es, sección de Calidad del Aire en Madrid

- Google:

  Google[14], por otra parte, ofrece una visualización sencilla como respuesta a preguntas relacionadas con la calidad del aire en tu región. Sin embargo, se trata exclusivamente de un índice de calidad al momento y no lo relaciona con ninguna variable externa, ni muestra un histórico, ni muestra el estado de otras ubicaciones cercanas. Los datos, que obtiene a través de la empresa israelí *BreezoMeter*[15], sólo tienen en cuenta seis partículas: monóxido de carbono (CO), óxidos de nitrógeno (NOx), partículas en suspensión (PM10 y PM2,5), ozono (O3) y dióxido de azufre (SO2). En la figura 4 podemos ver la interfaz descrita en búsquedas realizadas a través de móvil y escritorio.

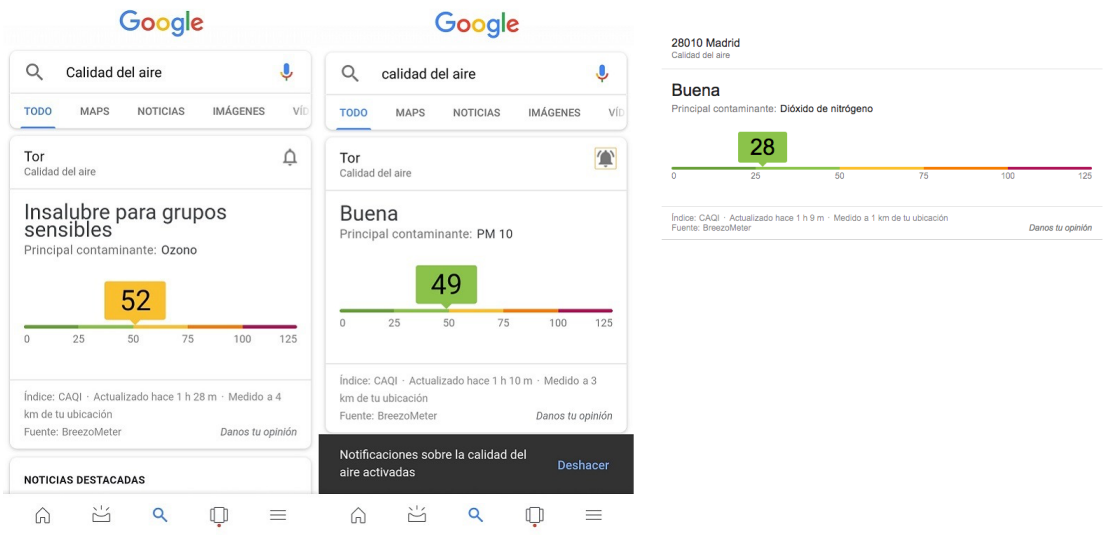

Figura 4: Detalle del *widget* de la Calidad del Aire de Google, en móvil (izquierda) y en escritorio (derecha)

- BreezoMeter:

  Esta empresa (que es la fuente de datos del *widget* de Google antes mencionado) ofrece una API para desarrolladores[15] con diversos datos acerca del Índice de Calidad del Aire. Además, en la propia web podemos ver un mapa de calor interactivo[16] con diversos datos sobre partículas contaminantes, así como otros factores influyentes en la salud como el polen. Aunque sí podemos ver en el *heatmap* (figura 5) los valores del ICA para una zona geográfica concreta (no sólo los puntos de medición), no tenemos otros factores como los que se plantean en este proyecto en relación con la calidad del aire.

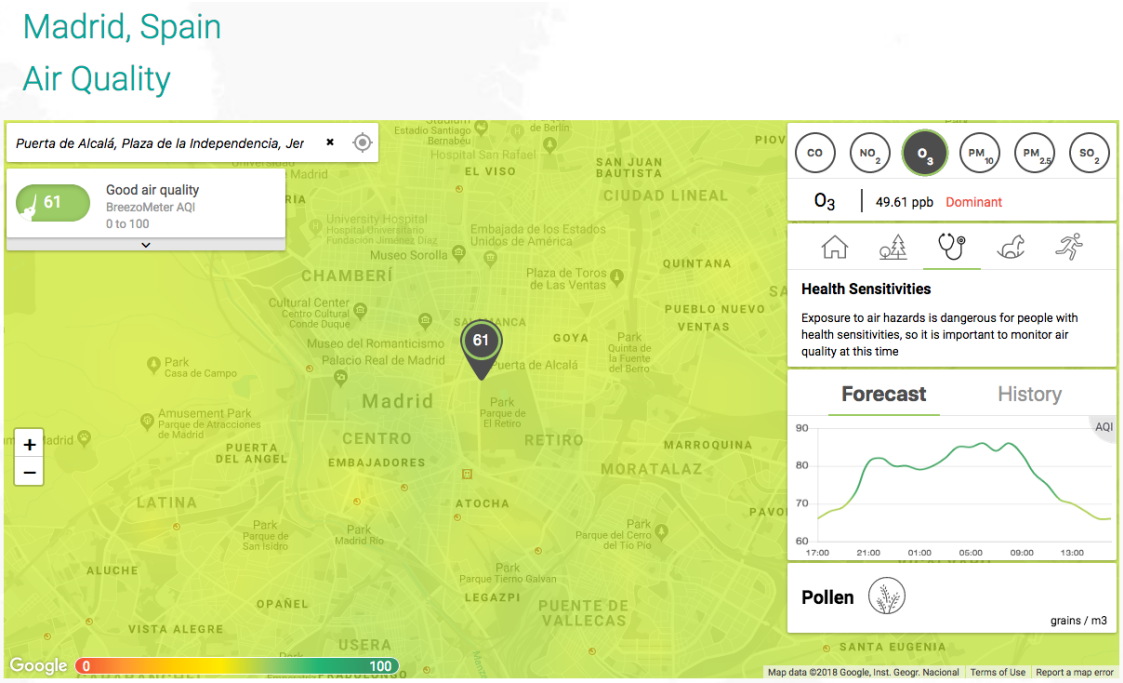

Figura 5: *BreezoMeter*: *Heatmap* de calidad del aire centrado en Madrid

- waqi o aqicn.org:

  El proyecto internacional "World Air Quality Index" (waqi.info o aqicn.org) muestra un mapa mundial de la calidad del aire en tiempo real, la posición geográfica de las estaciones de medición a nivel mundial, y una predicción de este índice para ciertas zonas de Asia. Como en otras alternativas mencionadas anteriormente, se puede consultar el gráfico con el histórico de los valores de las partículas, pero en relación con el trabajo aquí presentado faltaría obtener un valor del ICA para una zona dada, u obtener otros datos complementarios. En la figura 6 podemos ver la interfaz de la página web del proyecto, centrada en la ciudad de Madrid.

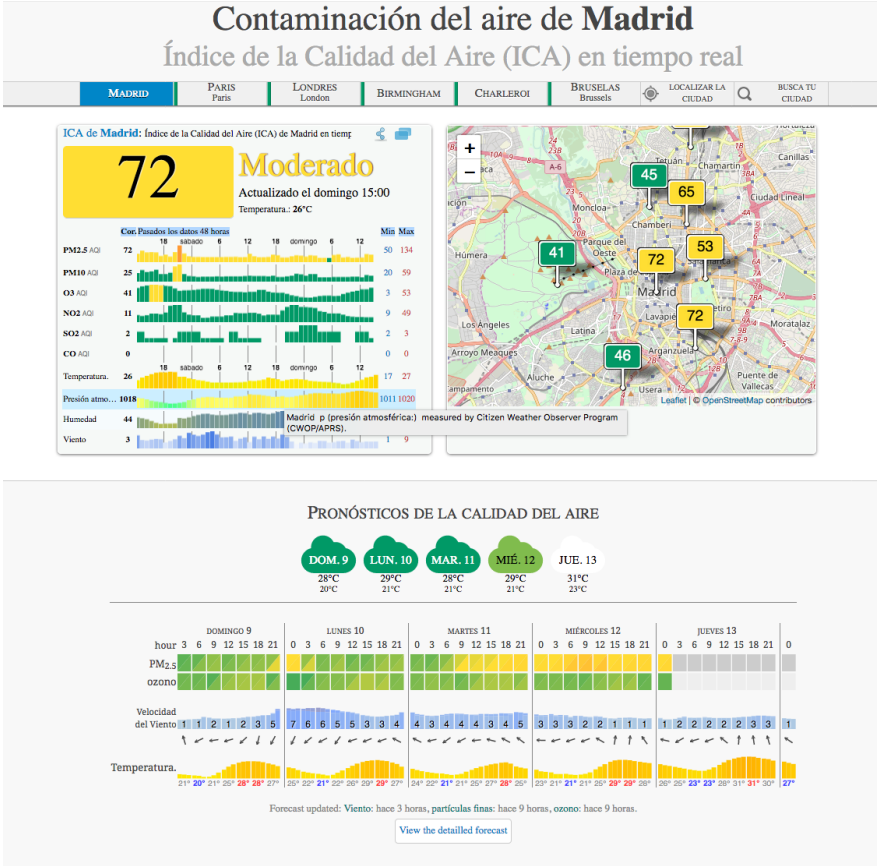

Figura 6: Página principal del proyecto *WAQI*, web de *aqicn*

2.3. Tecnologías utilizadas

En esta sección se detallan las tecnologías, librerías y componentes empleados en cada una de las diversas partes del trabajo desarrollado.

2.3.1. Listado de términos y tecnologías utilizadas

1. **Frontend**: En diseño de *software* el *frontend* (o *front-end*) es un tipo de abstracción para la capa de presentación, la parte del *software* que interactúa con los usuarios. La capa de *frontend* es responsable de recolectar los datos de entrada del usuario y transformarlos ajustándolos a las especificaciones que demanda el *backend* (capa de acceso a datos) para poder procesarlos. Es el equivalente a la interfaz del producto final.

2. **Backend**: es una abstracción para la capa de acceso a datos, que se encarga de procesar los datos del *frontend* (de entrada o de salida), devolviendo generalmente una respuesta que el *frontend* recibe y expone al usuario de una forma entendible para este. Es el equivalente al motor o esqueleto del producto final.

3. **Javascript** (abreviado comúnmente JS) es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

4. **ECMAScript**[17] es una especificación de lenguaje de programación Javascript publicada por *ECMA International*. El desarrollo empezó en 1996 y estuvo basado en el popular lenguaje JavaScript propuesto como estándar por Netscape Communications Corporation. Actualmente está aceptado como el estándar *ISO 16262*. ECMAScript define un lenguaje de tipos dinámicos ligeramente inspirado en Java y otros lenguajes del estilo de C. Soporta algunas características de la programación orientada a objetos mediante objetos basados en prototipos y pseudoclases. La mayoría de navegadores de Internet incluyen una implementación del estándar ECMAScript.

5. **Babel**[18] es una herramienta que compila (transpila) código escrito en versiones más

actuales de ECMAScript (2015+) y lo convierte a versiones de Javascript capaces de ejecutarse en entornos o navegadores antiguos.

6. **Node.js**[19] es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.

7. **npm**[20] es el manejador de paquetes por defecto para Node.js, un entorno de ejecución para JavaScript. Consiste en un cliente de línea de comandos (`npm`) y una base de datos online para paquetes públicos y privados, llamada *npm registry*.

8. **MongoDB**[21] es un sistema de base de datos NoSQL orientado a documentos, desarrollado bajo el concepto de código abierto.

9. **NoSQL**: en lugar de guardar los datos en tablas como se hace en las bases de datos relacionales, las bases de datos noSQL guardan estructuras de datos en documentos similares a JSON con un esquema dinámico (MongoDB utiliza una especificación llamada BSON), haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

10. **JSON**: acrónimo de JavaScript Object Notation, es un formato de texto ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript aunque hoy, debido a su amplia adopción como alternativa a XML, se considera un formato de lenguaje independiente. Una de las supuestas ventajas de JSON sobre XML como formato de intercambio de datos es que es mucho más sencillo escribir un analizador sintáctico (parser) de JSON.

11. **Eslint**[22] o JSLint es un analizador estático de código usado en el desarrollo de *software* para comprobar si un código escrito en Javascript se adecúa a las reglas de estilo del lenguaje.

12. **React**[23] (también llamada React.js o ReactJS) es una biblioteca Javascript de código abierto para crear interfaces de usuario con el objetivo de animar al desarrollo de apli-

caciones en una sola página. Es mantenido por Facebook, Instagram y una comunidad de desarrolladores independientes y compañías. Su objetivo es ser sencillo, declarativo y fácil de combinar.

13. **Turf**[24] es una librería de Javascript para análisis espacial (datos geo-localizados). Incluye operaciones espaciales, funciones para trabajar con datos GeoJSON, y herramientas de clasificación y análisis estadístico. Puede ser usado tanto en web como en servidor.

14. **GeoJSON** es un formato estándar abierto diseñado para representar elementos geográficos sencillos, junto con sus atributos no espaciales, basado en JSON. El formato es ampliamente utilizado en aplicaciones de cartografía en entornos web.

2.3.2. *Stack* tecnológico

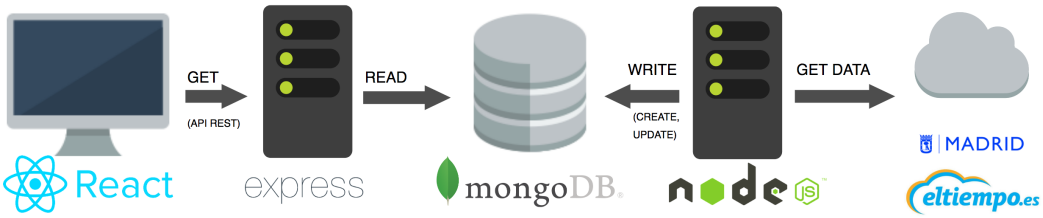

Figura 7: Diagrama de la arquitectura del proyecto

Para el desarrollo de la arquitectura propuesta, se ha decidido hacer uso del *stack* tecnológico conocido como MEAN, o su variante MERN [25]. MEAN (o MERN) es un conjunto de *frameworks fullstack* para la web compuesto por MongoDB, Express, Angular (o React en el caso de MERN) y Node. Este *stack* ofrece la posibilidad de integrar de forma rápida y robusta dichas tecnologías web, dándonos la posibilidad de desarrollar un producto completo en todas sus capas de aplicación.

2.3.3. Desarrollo de la web, entorno *front*

Como se ha comentado anteriormente, el *framework* utilizado para el desarrollo de la capa *front* de nuestra aplicación web es Reactjs.

2.3.4. Angular frente a React

React tiene varias ventajas y desventajas frente a su competidor directo Angular (ver MEAN vs. MERN).

1. **Ciclo de vida y desarrollo**

   Angular está desarrollado y mantenido por Google. Previamente llamado Angularjs, ha pasado por varias versiones incompatibles entre sí que introducían cambios considerables [26].

   React está desarrollado y mantenido por Facebook. De acuerdo con algunas fuentes, React está más extendido entre los productos de Facebook que Angular entre los de Google [27]. En los principios de diseño de React (por Facebook) se establece la estabilidad como valor fundamental a la hora de transicionar entre versiones [28]. Esto aporta más flexibilidad en el desarrollo y sobre todo en el mantenimiento del *software* a desarrollar.

2. **Componentes**

   Ambos *framework* usan componentes, si bien Angular está más basado en plantillas y lógica separadas, React trabaja más con componentes en bloque capaces de manejar información.

3. **Lenguaje**

   Angular usa Typescript, variedad de Javascript desarrollada por Microsoft que ofrece, entre otras opciones, tipado estático para las variables. La utilización de tipado estático ofrece robustez al código, pero añade mucha sobrecarga al proyecto [29].

   React usa ES6, variante de Javascript con influencias de Python y otros lenguajes, como la inclusión de programación funcional entre otros.

4. **Plantillas**

   Angular utiliza HTML con una sintaxis específica de Angular para elementos que son gestionados por la lógica del componente, por ejemplo, iteradores. React usa JSX, un

tipo de sintaxis de Javascript donde los objetos HTML con los que se trabajan se integran con la lógica de la aplicación.

5. **Estados y manejo dinámico de datos**

   Angular gestiona la variabilidad de las plantillas a través de variables implícitas en el HTML de la página que la lógica del componente se encarga de transformar. React utiliza un árbol de componentes con estados heredados (de forma opcional). Cuando un componente cambia su estado, se actualiza ese componente y los componentes que están por debajo en el árbol (aunque si no fuera necesario se puede actualizar sólo dicho componente).

   La diferencia fundamental radica en el "enlace de datos", o *data binding*. Este *data binding* puede ser unidireccional (React) o bidireccional (Angular). El enlace bidireccional de Angular cambia el estado del modelo cuando se actualiza la vista de la interfaz (por ejemplo, el usuario modifica un elemento interactivo como un campo de texto o un botón). El enlace unidireccional de React actualiza primero el modelo, y luego renderiza la vista (como hemos mencionado antes). En el caso de Angular el código es más limpio y más fácil de implementar, y en el caso de React el flujo de trabajo es mejor, ya que los datos fluyen en una sola dirección. Esto facilita el proceso de *debugging* del código.

6. **Flexibilidad y escalabilidad**

   Es fácil trabajar con React o con vuejs (más adelante) añadiendo librerías Javascript. Esto no es tan fácil en Angular por su uso de Typescript, ya que no es trivial integrar elementos de código en Javascript y elementos en Typescript sin transpilar primero a una base común.

Otras alternativas tecnológicas no utilizadas en este proyecto

1. **Vue (o vuejs)**[30]: se define como un "MVC rápido, intuitivo y reutilizable". Fue lanzado en 2014 por un exempleado de Google. Al igual que React, Vue utiliza un DOM virtual, trabaja con vistas por componentes reactivos, y se centra principalmente en la

librería principal, dejando de lado otras funcionalidades. Sin embargo, Vue tiene algunos problemas de integración en proyectos de gran volumen, y tanto la documentación como la comunidad existentes son menores que en el caso de React.

2. **Boostrap**: aunque es un *framework* más enfocado a estilos visuales, trabaja muy bien la parte *responsive* de una página web, tiene un soporte y una comunidad muy buena detrás, y es muy flexible. Sin embargo, también es un *framework* muy pesado que añade mucha sobrecarga al proyecto [31].

3. **Elm**[32] se define como un "lenguaje estable basado en dominio para crear *webapps* de forma declarativa". Es puramente funcional, y está desarrollado enfocado en la usabilidad, el rendimiento y la estabilidad. Tiene un compilador que trabaja con tipado estático, lo cual da mejor *feedback* al desarrollador que el código transpilado de React, pero no tiene tantos paquetes como React, no se ha lanzado aún una primera versión estable (o versión 1.0), y React se integra mejor con código existente.

2.3.5. Servidor de interfaz para la web

Dado que nuestra página web tiene que interactuar con nuestra base de datos, aunque sólo sea para operaciones de lectura, hace falta una capa de servidor que interprete las peticiones *CRUD*[33] (*Create, Read, Update and Delete*, en este caso sólo *Read*) y realice las *queries* (consultas) oportunas a la base de datos.

Para esta capa se ha decidido emplear express (express.js) como parte del *stack* mencionado anteriormente (MERN). Express (o Expressjs) es el paquete de Nodejs con más estrellas ("favoritos") de la comunidad en Github. El creador original del proyecto, TJ Holowaychuk, lo vendió a una *startup* llamada StrongLoop, que más tarde fue adquirida por IBM. Sin embargo, el proyecto de Express ha seguido un desarrollo paralelo de la mano de Doug Wilson, uno de los desarrolladores originales del proyecto [34].

Express se define como un *framework* de aplicaciones web minimalista y flexible para Nodejs, que provee de un conjunto de características para aplicaciones web y móviles. La ventaja principal de Express radica en su simpleza y facilidad de integración en proyectos de Javas-

cript, la desventaja principal radica en el número de dependencias asociadas al paquete, como es habitual en este tipo de proyectos (debido a los *node_ modules* o dependencias anidadas en proyectos de Javascript).

Por poner un ejemplo, en la figura 8 tenemos el mismo modelo de ruta implementada en Express y en otro *framework* alternativo en Javascript, Fastify.

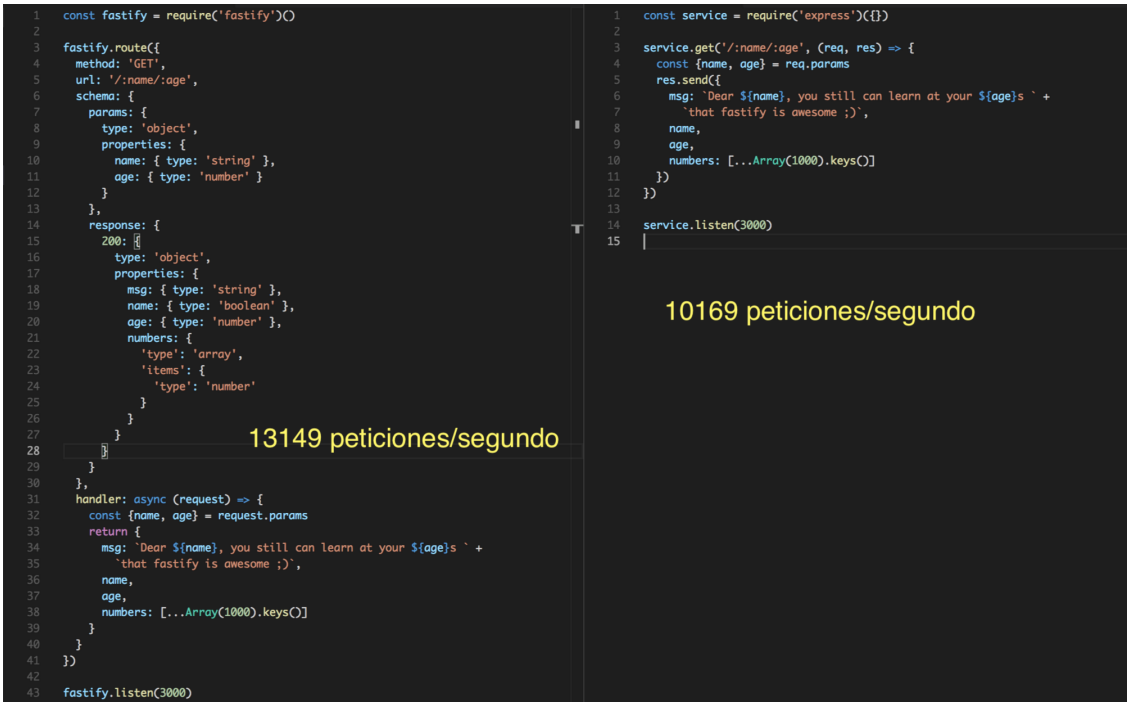

Figura 8: Comparativa del mismo código usando Fastify (izq.) y Express (derecha) en peticiones/segundo

A pesar del peor rendimiento, dado que nuestro proyecto tendrá un número muy bajo de *endpoints* o rutas, y es más importante la simpleza del código y su integración con la base de datos, se decidió usar Express [35].

Otras alternativas para esta capa del *stack*:

1. **Ruby on Rails** [36], *framework* de servidor en Ruby con una gran comunidad detrás. La principal desventaja frente a Express hubiera sido emplear un lenguaje de programación diferente al del resto del proyecto.

2. **Django** [37], *framework* de servidor en Python. Este *framework* es muy popular cuando el resto del proyecto está desarrollado también en Python (como *Flasjk*). Como no es el caso, se rechazó.

2.3.6. Base de Datos

Antes de hablar de la elección tomada en esta capa del *stack* tecnológico, es necesario hablar de bases de datos relacionales y bases de datos no relacionales. Las bases de datos SQL suelen ser bases de datos relacionales (RDBMS), y las NoSQL se conocen como no-relacionales o distribuidas.

Las bases de datos SQL están basadas en tablas, mientras que las bases de datos noSQL están basadas en documentos, pares clave-valor, grafos, etc. Esto hace que las BBDD SQL tengan esquemas predefinidos, mientras que las NoSQL tengan esquemas dinámicos para datos no-estructurados.

Para *queries* complejas las bases de datos SQL son mejor opción que las bases de datos NoSQL, ya que la interfaz de *queries* disponible en BBDD NoSQL no tiene la variedad de operaciones que tienen las bases de datos SQL en cuanto a las *queries* a realizar. Esto quiere decir que en función de las operaciones que se deseen realizar sobre la base de datos, puede que algunas no sean realizables con bases de datos NoSQL (o que la manera de realizarlas sea mucho más compleja).

En cuanto al tipo de datos a almacenar, las bases de datos NoSQL funcionan mejor que las BBDD SQL para datos jerárquicos gracias al sistema de declaración por "objetos" (que permiten anidado). Por otra parte, las BBDD SQL soportan mayor carga transaccional que las BBDD NoSQL.

Algunos ejemplos de bases de datos SQL serían MySql, Oracle, Sqlite, Postgres o MS-SQL, y algunos ejemplos de bases de datos NoSQL serían MongoDB, BigTable, Redis, RavenDb, Cassandra, Hbase, Neo4J o CouchDb.

Dentro de las bases de datos NoSQL hay diversos tipos que no tendría sentido aplicar en este proyecto por la naturaleza de los datos (por ejemplo BBDD orientadas a grafos como

Neo4J). Sin embargo, sí se considera relevante esta comparativa entre las BBDD SQL y NoSQL porque fue uno de los aspectos que más se tuvo en cuenta a la hora de diseñar la arquitectura de la base de datos del proyecto.

Al final se optó por MongoDB por las siguientes razones:

1. La estructura de los datos no es relacional: tenemos una tabla o documento (relación entre términos SQL y MongoDB en [38]) por cada zona del mapa y no hay relaciones entre sí.

2. Las *queries* a realizar no son excesivamente complejas, ya que nos interesa obtener datos sin uniones ni búsquedas demasiado avanzadas.

3. La integración con el *stack* utilizado (MERN), como ya se ha comentado anteriormente, era otro punto a favor.

4. Las *queries* siguen una sintaxis mucho más simple, directa, y similar a la propia sintaxis del lenguaje utilizado (Javascript), lo cual se valoró positivamente.

2.3.7. *Scripts* de recolección/procesamiento de datos

Inicialmente el proyecto se planteó como una serie de *scripts* de recolección de datos integrados con un *bot* de Telegram, que darían la información bajo demanda del usuario acerca del estado de la contaminación del aire. Telegram es una plataforma de mensajería instantánea multiplataforma con una API de desarrollo que permite, entre otras cosas, la creación de *bots* o agentes conversacionales con los que interactúa el usuario, ya sea directa (por conversación) o indirectamente (a través de tareas en grupos con más personas).

Aunque esto no se llevó a cabo, los *scripts* originales estaban desarrollados en Nodejs (y algunos en Python).

Nodejs[19] es un entorno de ejecución *open-source* multiplataforma de Javascript que ejecuta código Javascript fuera del navegador. Para esta capa del *stack*, los requisitos definidos en la creación del proyecto eran:

1. Que los *scripts* pudieran conectarse con servidores públicos y recopilar los datos con los que trabaja este proyecto.

2. Que el lenguaje empleado tuviera las librerías necesarias para procesar los datos y operar con ellos.

3. Que los *scripts* tuviesen una estructura modular e integrable con la base de datos para guardar los datos recolectados.

Para el primer requisito, hubiera valido casi cualquier lenguaje. Se optó por Javascript y, por lo tanto, Nodejs, al definir el resto del *stack* y las otras capas a implementar.

Para el segundo requisito se usó inicialmente Python[39], que posee mejores herramientas para el tratamiento de archivos .csv[40] y similares, para un mejor procesamiento de los datos. Sin embargo, por unificar los diferentes módulos de la aplicación y tener una mejor integración, se acabaron desarrollando todos en Javascript.

En el caso del último requisito, la base de datos empleada (mongoDB) tiene librerías e integraciones con los lenguajes C, C++, C#, Java, Javascript (Nodejs), Perl, PHP, Python, Ruby, y Scala. No hubiera sido problema integrar la base de datos con una aplicación desarrollada en la mayoría de estos lenguajes. Sin embargo, de nuevo se decidió utilizar Javascript para unificar más el proceso de desarrollo.

El punto de entrada del flujo desarrollado para la recolección de datos es el archivo `crawler.js`, que importa el resto de *scripts* (`particles`,`meteo` y `traffic`). Cada uno de los otros *scripts* recolectan los datos de forma asíncrona y envían los resultados a través de una función *callback*. El proceso principal del *crawler* ("programa que busca información en la web y la indexa para su posterior uso") es el que se encarga de ir llamando al resto de métodos progresivamente trabajando con los *callbacks* enviados a las funciones. Este script se conecta a la base de datos en mongoDB y procesa los datos enviados por cada una de las funciones. Cuando ya tiene todos los datos, ejecuta las operaciones de escritura y actualización sobre la base de datos. Este proceso se ejecuta con un planificador en forma de *cron* gracias a una

librería de Javascript que emula la función *cron* del Sistema Operativo con el objetivo de realizar este proceso de recolección de datos una vez cada hora, 24 horas al día.

3. Diseño de la solución

Esta sección del documento describe el sistema de diseño básico para el *software* desarrollado para el proyecto. Se estructura a su vez en tres subsecciones: análisis, diseño e implementación.

3.1. Análisis

En esta subsección se definen el alcance del trabajo, sus funcionalidades y límites, los requisitos que definirán la posterior fase de diseño, y las restricciones y límites que apliquen al proyecto.

3.1.1. Alcance del trabajo

Como se ha dicho anteriormente, este proyecto consiste en el diseño, implementación y evaluación de una plataforma web para el análisis visual de los factores que contribuyen potencialmente a la calidad del aire en la ciudad de Madrid, así como el conjunto de servicios relacionados para hacer funcionar dicha plataforma.

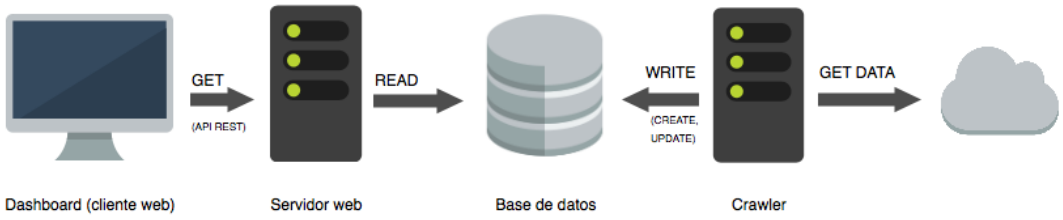

Figura 9: Diagrama de componentes del proyecto

En la figura 9 podemos ver la correspondencia del diagrama del *stack* tecnológico con la arquitectura de la plataforma web. Como se puede ver en este esquema, la plataforma web está constituida por:

1. Un *dashboard* o tablero donde se puede consultar la información relativa a estos datos, separados en 24 sectores en los que está dividida la información en la ciudad, delimi-

tados por un mapeo de *Voronoi* sobre las estaciones de calidad del aire repartidas por Madrid. En este tablero disponemos de información tal como un mapa de la ciudad dividido en estos 24 sectores y gráficas individuales para cada sector donde podemos observar distintas variables significativas en este análisis. En este detalle tenemos datos como la calidad general del aire, factores meteorológicos, o la intensidad media del tráfico para la zona dada, representado en series temporales para las últimas 24 h, última semana, etc.

2. Un servidor web en la misma máquina donde se encuentra la base de datos, sirviendo una serie de rutas específicas para la web que operan de forma concreta y definida sobre la base de datos.

3. Una base de datos que interactúa con el *crawler* para la escritura y actualización de los datos presentes en ella, con el objetivo de aligerar la carga de consultas sobre los servidores de los datos originales (que a menudo disponen de *rate limits*, o límites de consulta por hora). Esta base de datos interactúa con la web a través de un servidor intermedio que recibe las peticiones de la web y realiza las consultas oportunas a esta base de datos.

4. Un *crawler* o recolector de datos formado por un conjunto de *scripts* que recopila datos públicos de diferentes fuentes con el objetivo de proveer al *dashboard* con un histórico de datos que va recogiendo. Estos datos se almacenan en una base de datos para su posterior consulta.

Como tal, los requisitos definidos inicialmente en la creación del proyecto, enumerados de manera informal, serían:

- Se requiere representar una gran cantidad de factores que pueden contribuir a la calidad del aire en la ciudad de Madrid.
- Se sugiere la posibilidad de disponer de varios datos de interés que se puedan correlar o representar visualmente.

- La web ha de ser accesible y funcional, con un diseño adaptado a diversas plataformas y que pueda ser consultado con facilidad independientemente del entorno y del usuario.
- Todas las capas del proyecto (detalladas más adelante) han de poder integrarse con facilidad, con el objetivo de simplificar el proceso de desarrollo del proyecto.

Los aspectos que quedan fuera del alcance de este proyecto son:

- Los datos que mostrará la web nunca serán en tiempo real, ya que en primer lugar la recolección de datos se realiza una vez cada hora, y en segundo lugar los datos públicos de los que se dispone tienen un retraso de una o dos horas con respecto a la hora actual de la consulta.
- No se realizarán predicciones o estimaciones sobre datos futuros, mediante predicción de series temporales ni ninguna técnica de aprendizaje automático sobre los datos.
- No se harán estudios estadísticos de ningún tipo sobre la correlación entre las variables incluidas en las visualizaciones de datos en este proyecto.
- Esta plataforma será únicamente con fines informativos, no se avanzará en ningún aspecto relacionado con la salud ni se dará ningún tipo de indicación para el usuario en dichos aspectos.

3.1.2. Metodología de trabajo

En esta subsección se detalla la metodología de investigación y análisis que se ha seguido para la realización de este proyecto.

Para la fase de análisis, recogida de requisitos y diseño se ha trabajado con las primeras fases del modelo de diseño de doble rombo, de Design Council[41][42], el cual podemos ver detallado en la figura 10 Este modelo describe cuatro fases por las que debe de pasar un producto: descubrimiento, definición, desarrollo y entrega. La investigación se integra en todo el proceso, perfilando las mejores soluciones e iterando sobre éstas.

Las fases del modelo se pueden resumir de la siguiente manera:

1. Descubrimiento: en esta fase se analizan problemas y se identifican oportunidades.

2. Definición: en esta fase se alinean las ideas generales de la fase anterior con los objetivos marcados.

3. Desarrollo: los conceptos de diseño se prueban en prototipos para asegurar que cumplen con los objetivos previamente dichos.

4. Entrega: evaluación del producto final por parte del usuario.

En este proyecto se han trabajado sobre todo con las dos primeras fases para la definición de requisitos en la investigación con usuarios, ya que la fase de desarrollo o prototipado se ha hecho principalmente desarrollando el producto final con código. Por otro lado, la parte de entrega puede verse reflejada más adelante en el apartado de pruebas y conclusiones.

La adaptación de las fases anteriormente mencionadas al presente proyecto quedaría de la siguiente manera:

1. Recogida de requisitos (sección 3.2): los requisitos obtenidos en la fase de investigación obvian cualquier aspecto de diseño de la arquitectura ya realizado en el proyecto, y se centran en las necesidades de los usuarios con respecto a la aplicación. Para la recogida de requisitos se ha realizado investigación mediante observación y entrevistas con posibles usuarios de la plataforma.

2. Análisis: para la fase de definición o análisis se han cruzado los requisitos obtenidos de los usuarios con el alcance del proyecto, los recursos disponibles (fuentes de los datos, material y tiempo) y el objetivo general del proyecto. Esta fase se ve muy reflejada en el apartado de alcance del proyecto.

3. Diseño, prototipos (sección 3.3): a pesar de no dar mucho peso a la fase de desarrollo a través de diseño, sí se ha realizado un diseño base de la aplicación antes de la fase de desarrollo del *software* para orientar tanto el aspecto visual de la aplicación como la funcionalidad de sus componentes. En esta fase se volvió a divergir un poco más, siguiendo el modelo del doble rombo propuesto, ya que muchas ideas quedarían descartadas en la fase de desarrollo sobre código.

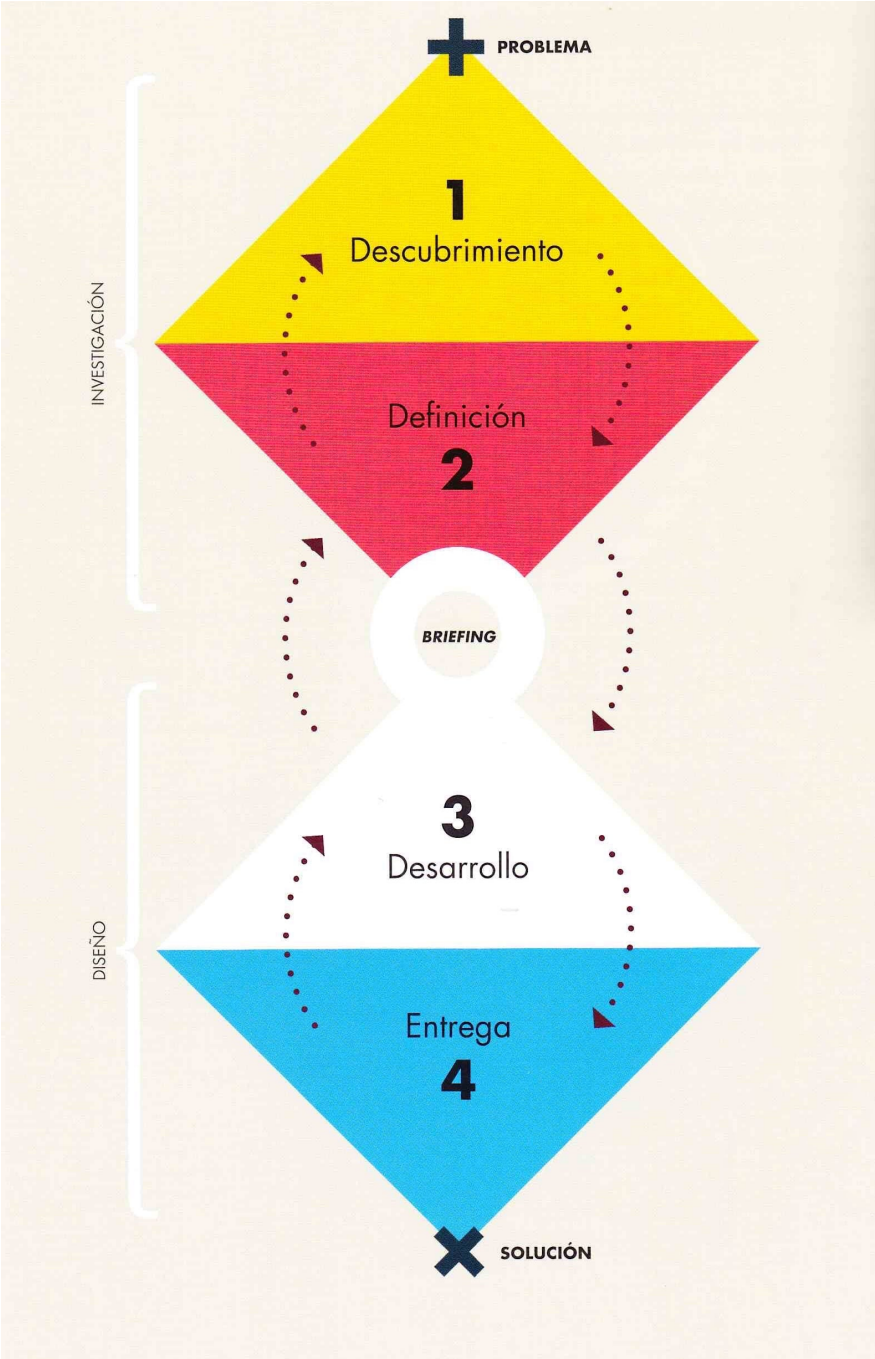

Figura 10: Modelo del doble rombo, Design Council. Fuente: Blume [43]

3.2. Recogida de requisitos

En este apartado se describen los requisitos recogidos tal y como se describe en la metodología de trabajo (sección 3.1.2).

3.2.1. Requisitos

Los requisitos definen el entorno de trabajo, las funcionalidades y las interacciones del usuario con el sistema desarrollado. Cada requisito está representado con los siguientes campos:

1. ID: compuesto por las iniciales RS (Requisito del Sistema) seguido del número de requisito correspondiente.
2. Título: un título breve del requisito correspondiente.
3. Descripción: una explicación más elaborada del requisito.
4. Prioridad: niveles de prioridad del requisito. Valores: baja, media o alta.

ID	RS-01
Título	Sistema Operativo para desarrollo
Descripción	Los sistemas operativos utilizados en desarrollo serán Ubuntu Linux 18.04 y macOS 10.13, ambos basados en Unix.
Prioridad	Media

Tabla 1: Requisito 01 - Sistema Operativo para desarrollo

ID	RS-02
Título	Comunicación de la infraestructura
Descripción	Todos los componentes del proyecto (la web, el servidor web-BBDD, la base de datos, y los *scripts* de recogida de datos) deben de poder comunicarse entre sí con el fin de leer o escribir información.
Prioridad	Alta

Tabla 2: Requisito 02 - Comunicación de la infraestructura

ID	RS-03
Título	Acceso a la plataforma
Descripción	El usuario ha de poder acceder a la aplicación a través de una web de manera sencilla y desde cualquier plataforma.
Prioridad	Alta

Tabla 3: Requisito 03 - Acceso a la plataforma

ID	RS-04
Título	Recogida de datos
Descripción	Se necesita recoger la mayor cantidad de datos públicos relevantes sobre las variables a analizar a lo largo del tiempo, y se deben almacenar estos datos para su posterior consulta.
Prioridad	Alta

Tabla 4: Requisito 04 - Recogida de datos

ID	RS-05
Título	Coordenadas geográficas
Descripción	Todos los datos geoposicionados deben de usar el mismo sistema de coordenadas para una mejor integración entre las fuentes de los datos.
Prioridad	Media

Tabla 5: Requisito 05 - Coordenadas geográficas

ID	RS-06
Título	Interacción del usuario con la UI
Descripción	El usuario deberá de poder interactura con la UI, seleccionando la zona de la que desea consultar la información
Prioridad	Alta

Tabla 6: Requisito 06 - Interacción del usuario con la UI

ID	RS-07
Título	Accesibilidad en la interfaz
Descripción	El usuario debe de ser capaz de consultar los datos en la interfaz aunque disponga de algún tipo de impedimento visual
Prioridad	Media

Tabla 7: Requisito 07 - Accesibilidad en la interfaz

ID	RS-08
Título	Adaptación a datos incompletos
Descripción	El sistema deberá de poder mostrar los últimos datos disponibles, aunque estos estén incompletos en la base de datos
Prioridad	Alta

Tabla 8: Requisito 08 - Adaptación a datos incompletos

ID	RS-09
Título	Tiempo en línea
Descripción	Todos los módulos del sistema deberán estar activos 24 horas al día con el objetivo de recolectar datos y servirlos a través de la web al usuario
Prioridad	Alta

Tabla 9: Requisito 09 - Tiempo en línea

ID	RS-10
Título	Idioma de la interfaz
Descripción	El idioma utilizado en la interfaz será el castellano
Prioridad	Baja

Tabla 10: Requisito 10 - Idioma de la interfaz

ID	RS-11
Título	Estructura de la interfaz
Descripción	La interfaz debe de poder mostrar el mapa de las zonas, el sistema de tarjetas informativas, y la gráfica de variables
Prioridad	Media

Tabla 11: Requisito 11 - Estructura de la interfaz

3.2.2. Restricciones

- Los datos presentados, por restricciones de la plataforma de datos original, tienen un retraso de entre una y dos horas.
- Aunque no se suele dar este caso, estos datos, pueden presentar carencias debido a algún fallo de la plataforma de datos original o del servicio de recogida de datos (como, por ejemplo, filas incompletas o inexistentes para algunas horas).
- La plataforma se desarrollará exclusivamente en castellano, lo cual dificulta la comprensión de los textos que aparecen a los usuarios que desconozcan este idioma. Sin embargo, se sobreentiende que al ser un producto destinado para los usuarios del área metropolitana de Madrid, la mayoría de dichos usuarios serán castellanoparlantes.
- La plataforma presentará una interacción limitada con los datos mostrados, sin entrar en más detalle ni ofrecer relaciones entre las variables u otras características de posible valor.

3.2.3. Especificaciones y casos de uso

En este apartado se detallan las funcionalidades que tendrá la aplicación, así como las tareas que pueden realizar los usuarios a través de la aplicación. También se muestran diagramas de uso para cada posible usuario.

No se incluyen los casos de uso para usuarios administradores del sistema (por ejemplo, el servidor web o la base de datos) por ser casos de uso implícitos en el proyecto, aunque sí se detallan más adelante.

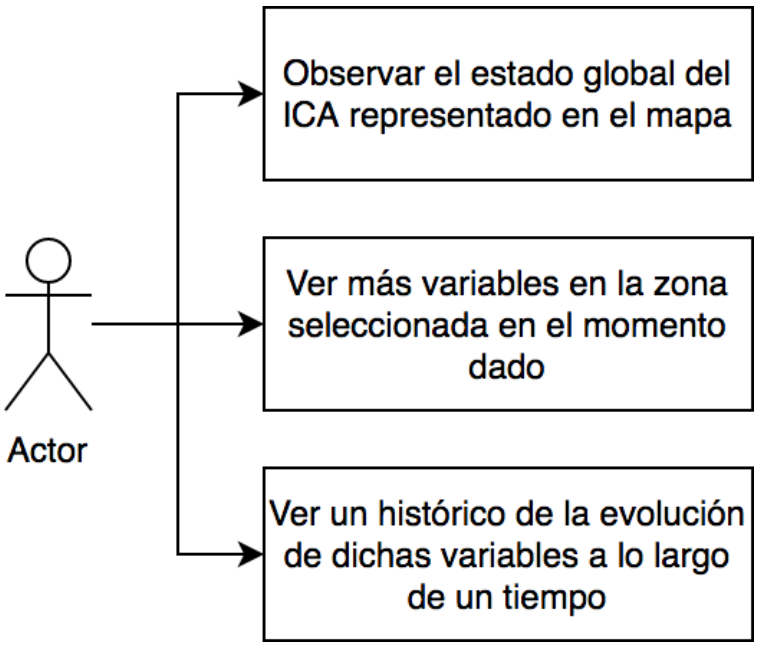

Figura 11: Casos de uso genéricos para un usuario de la aplicación

Para las tablas de los casos de uso enumerados a continuación se ha seguido la siguiente estructura:

1. ID: compuesto por las iniciales CU (Caso de Uso) seguido del número de caso de uso correspondiente.
2. Título: un título breve del caso de uso correspondiente.
3. Precondición: una explicación del disparador del evento asociado al caso de uso, normalmente la acción realizada por el usuario.
4. Postcondición: una explicación del resultado o respuesta del sistema ante dicha acción.

ID	CU-01
Título	El usuario abre la UI para consultar el mapa
Precondición	El usuario accede a la página web del proyecto a través de la URL
Postcondición	El servidor donde está hosteada la página web recibe la petición y sirve los archivos HTML, CSS y JS precompilados de la interfaz. Estos hacen una petición al servidor express de la aplicación, el cual devuelve el estado del mapa tras consultar a la base de datos.

Tabla 12: Caso de uso 01 - El usuario abre la UI para consultar el mapa

ID	CU-02
Título	El usuario selecciona una zona
Precondición	El usuario selecciona una zona para obtener más información sobre ella
Postcondición	El componente panel se refresca mostrando los datos previamente guardados relativos únicamente a la zona seleccionada.

Tabla 13: Caso de uso 02 - El usuario selecciona una zona

ID	CU-03
Título	El usuario desea consultar más información acerca de los valores mostrados
Precondición	El usuario selecciona la opción de mostrar más información presente en el componente panel
Postcondición	La aplicación hace una petición específica sobre la zona dada dentro de una ventana de tiempo, y muestra los datos recibidos en forma de gráfico temporal

Tabla 14: Caso de uso 03 - El usuario desea consultar más información acerca de los valores mostrados

3.3. Diseño

En esta subsección se detalla el diseño de la arquitectura a implementar en los distintos módulos incluidos en la aplicación.

En primer lugar, se describe el proceso de investigación, análisis y diseño de la aplicación. A continuación, se explica en detalle las decisiones tomadas acerca del entorno de la aplicación, las decisiones de diseño de arquitectura, y por último una descripción en detalle del sistema por módulos en términos de sus interfaces y dependencias con otros componentes para facilitar su integración futura con otros trabajos.

3.3.1. Diseño de la interfaz web

En este punto se cubre la fase de diseño conceptual y visual de la interfaz web que se presenta en este proyecto. Como se puede ver en detalle en la sección de metodología (3.1.2), después de la recogida de requisitos ya se estudió que el panel o *dashboard* que componía la interfaz tenía que mostrar una serie de elementos, colocados de distintas formas en función del dispositivo desde el que se consultase esta web.

- Diseño del logo: para el logo se tomó la decisión de realizar un diseño simple, basado en un logotipo sencillo y utilizando el nombre del proyecto. Después de desechar otros nombres (como "airmon") se escogió "**airmad**", cuyas letras proporcionaban un impacto visual sencillo, claro e informativamente directo. Para que el logotipo funcionase mejor en web bajo distintas resoluciones y densidades de píxeles, se decidió usar un archivo vectorial *.svg* con el texto, así como una fuente de uso público llamada Lobster. Lobster es una de las fuentes que se pueden encontrar, descargar y usar en Google Fonts, librería de catálogo tipográfico con más de 800 fuentes licenciadas para su uso libre, y APIs para usar dichas fuentes tanto en web (CSS) como en Android.

- Diseño de la UI: para la colocación de los elementos sobre la UI se tuvo en cuenta el tamaño de cada uno de los elementos: el mapa, la información del interior del panel, y las gráficas. En la fase de investigación se hizo un primer prototipo del aspecto de la

Figura 12: Versiones del logo: versión final arriba y alternativa abajo

web en baja fidelidad, y luego se hicieron algunas de pruebas de diseño visual para ver los elementos de su aspecto en alta fidelidad.

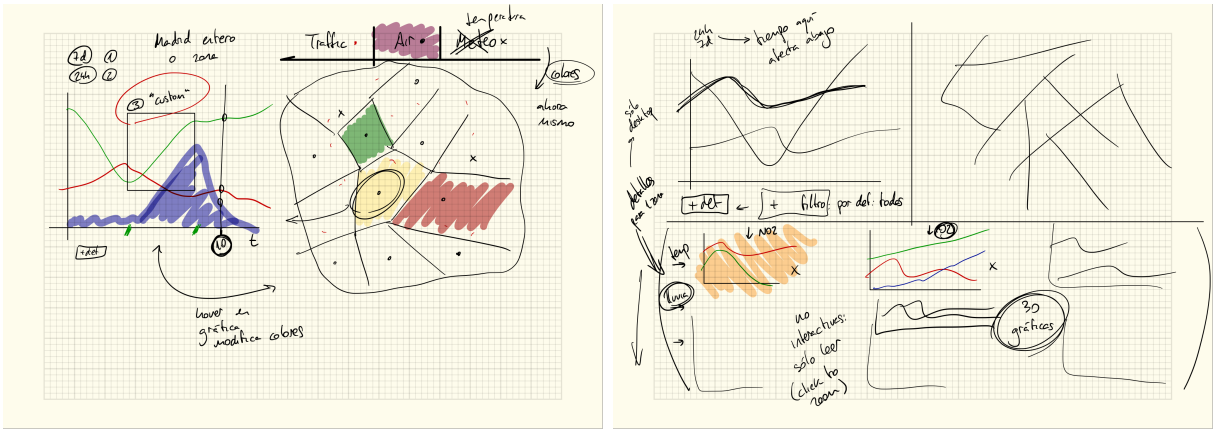

Figura 13: Wireframes de la interfaz a muy bajo nivel, utilizados en la fase de investigación

En las primeras fases de diseño de la interfaz se plantearon funcionalidades que no se han incluido en la versión final del proyecto, algunas por exigir una excesiva dedicación en relación con el resultado obtenido, y otras desechadas en fases posteriores de definición y rediseño.

Figura 14: Primera aproximación al diseño visual de los elementos de la web, en versión de móvil y tablet

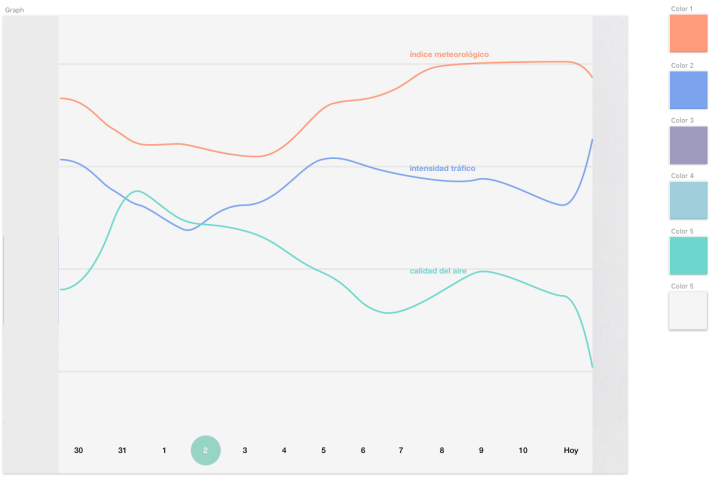

Figura 15: Diseño visual más avanzado del componente de gráficas

Figura 16: Detalle del diseño visual de las tarjetas de información mostradas en el componente de panel

3.3.2. Estudio del entorno

Como ya se ha mencionado antes, el proyecto está dividido en varias partes que actúan como las capas de aplicación de éste. Por lo tanto, se han tenido que llevar a cabo diversas decisiones, desde decisiones de arquitectura y desarrollo hasta decisiones del entorno de producción.

Esta sección detalla no tanto las tecnologías empleadas (detalladas en el apartado 2.3) como las decisiones llevadas a cabo para elegir una u otra tecnología en función de la situación y del contexto o entorno.

3.3.2.1 Interfaz web Como se ha contado anteriormente, se han tomado varias decisiones de cara a la implementación de este componente que han afectado en gran medida a su aspecto final.

En primer lugar, se decidió qué información debería mostrar el componente. Para esto se hizo una investigación de los datos de los que se podría disponer, revisando varias fuentes

de datos abiertos. Por otra parte, se hizo una serie de entrevistas a usuarios promedio para estudiar qué datos les gustaría encontrar en la plataforma y de qué manera. Se llegó a la conclusión de que, además del propio Índice de Calidad del Aire, sería interesante mostrar también otras variables para ver la relación entre el ICA y estas variables, detectar patrones de forma visual y entender mejor la evolución de los datos.

De cara a la implementación, se consideraron también varias librerías para desarrollo web, como se detalla en el apartado de Tecnologías utilizadas.

Se llevaron a cabo los siguientes niveles de decisión:

- A nivel de lenguaje: se consideró implementar este módulo usando Python u otros lenguajes de *scripting* con librerías para el desarrollo web. Por familiaridad con el lenguaje y por mejor adecuación de este para la tarea a realizar, se optó finalmente por Javascript.

- A nivel de librerías: se consideró usar varios *frameworks* anteriormente mencionados como Angular, React o Vue.js. Al final se optó por React por las razones anteriormente mencionadas (familiaridad, simpleza, facilidad de mantenimiento y mejor modularización de los archivos).

- Sobre otras tecnologías: React se puede usar en conjunto con otras tecnologías que pueden facilitar el desarrollo del proyecto. Librerías como *Redux* o *Mobx* ayudan a gestionar el estado de la aplicación con más profundidad que el propio *state* nativo que ofrece React. Estándares como *SASS* o *SCSS* ayudan a la gestión de hojas de estilos para la aplicación, permitiendo modularizar el estilo junto a los componentes usados, o incluso simplificar el código escrito (no el transpilado final) mediante sintaxis más limpia y útil.

  Si bien todas estas tecnologías son de gran utilidad en proyectos de gran envergadura, estrictamente hablando la parte de componentes de la interfaz del proyecto desarrollado se compone de tres módulos detallados más adelante. La simpleza de estos módulos a nivel de lógica y código llevaron a la conclusión de que era mejor no usar este tipo de

librerías para no añadir *overhead* o sobrecarga a los archivos finales generados para el navegador, que impactarían directamente en los tiempos de carga de la web.

3.3.2.2. Servidor HTTP(S) para la web Al igual que con el apartado anterior, antes de la fase de implementación de este módulo, se tuvo en cuenta los requisitos de este de cara a la integración con el resto del proyecto. Como hemos visto en el apartado de requisitos, lo más fundamental era:

1. El servidor tenía que recibir las peticiones HTTP(S) de la web y ofrecer una API REST para el intercambio de datos con la misma.
2. El servidor tenía que poder acceder a la base de datos para poder realizar todas las consultas que se presentaban en la API.

Por lo tanto, y como se detalló en el apartado de tecnologías utilizadas, se podrían haber utilizado diversas alternativas para esta capa que fuesen capaces de montar una API REST disponible para la web.

Los lenguajes con los que se integran las librerías de la base de datos (MongoDB) hubieran restringido esta capa a usar lenguajes como PHP, Ruby, Python o Javascript (Node y Express), pero se decidió utilizar este último para simplificar el proceso de implementar esta capa en la máquina de producción (usa el mismo entorno que las otras capas, Node).

Dentro de esta capa también teníamos la posibilidad de utilizar la librería nativa de MongoDB para Nodejs, o utilizar una abstracción de esta librería en forma de otra librería llamada Mongoose. Mongoose es una librería que se define como "una herramienta de modelado de objetos mongoDB diseñada para trabajar en entornos asíncronos" [44].

Mongoose permite definir esquemas más elaborados pero, de nuevo, no era necesario operar con datos de manera tan compleja, así que se optó por trabajar directamente con MongoDB a través de *queries* similares a las empleadas en los otros scripts detallados más adelante.

3.3.2.3. Base de datos Como ya se ha mencionado anteriormente, en este caso la principal decisión de diseño consistía en emplear una base de datos relacional o una no-relacional. Dentro de las bases de datos no-relacionales, la decisión es trivial, ya que depende del tipo de datos con el que se trabaja.

En nuestro caso, nuestro esquema de base de datos:

- Usa datos no-relacionales, sin dependencias entre las tablas (o documentos).
- Usa queries sencillas, sin uniones ni operaciones complejas.
- Usa datos no jerarquizados, basados en series temporales, y sin ningún esquema especial que no sea una tabla (por ejemplo, grafos).

Por todo esto, vemos que la mejor decisión dentro de la arquitectura (y dentro del *stack* tecnológico utilizado) es usar una base de datos no relacional basada en documentos como mongoDB.

Para poder acceder a la base de datos contamos con la librería de mongoDB mencionada anteriormente para Node, así como una interfaz gráfica llamada **mongoDB Compass**, que nos permite revisar todos los documentos guardados en la base de datos en forma de tabla, y comprobar el correcto funcionamiento de la aplicación.

3.3.2.4. Scripts de recogida de datos Como se detalló en el apartado de Tecnologías utilizadas, esta parte del proyecto es la más antigua en términos de desarrollo. Inicialmente se planteó que la interfaz del servicio de cara al usuario fuese una interfaz conversacional en forma de *bot* de Telegram.

Esta interfaz conversacional recibía del usuario una petición inicial indicando unas coordenadas geográficas, y se programaba para mandar un mensaje cada hora al usuario con los datos extraídos de diversas fuentes públicas acerca de la contaminación del aire y otras variables.

Al plantear este proyecto como Trabajo de Fin de Grado, se decidió adaptar el conjunto de *scripts* de recogida de datos para los requisitos del nuevo proyecto. Como tal, se definieron

unos estándares de calidad del código y se estandarizaron las interfaces de los módulos de cara a su posterior integración con un *script* final que aglutina todos los datos, los prepara, y los escribe en la base de datos.

Dado que la librería original que usaba la API de Telegram estaba disponible para el lenguaje Javascript (a través de Nodejs), los *scripts* principales como el de calidad del aire o el de tráfico están escritos también en Javascript siguiendo el estándar ES6 bajo la versión de Nodejs 10.10. El módulo relativo a las condiciones meteorológicas estaba escrito inicialmente en Python, pero se pasó a Javascript respetando las especificaciones de entrada y salida de datos. De esta manera, todos los submódulos de esta capa del proyecto se pueden importar como pseudo-librerías en el *wrapper* o módulo principal (también llamado *crawler*). Esto permite la separación de la lógica de este componente a través de las fuentes de datos originales, facilitando por lo tanto la labor de mantenimiento e integración del código en un futuro.

3.3.2.5. *Hosting* y despliegue de la aplicación A la hora de montar la infraestructura de la aplicación, hay una serie de requisitos que se aplican. El principal requisito en este caso era el tiempo en línea del sistema, el cual necesita estar disponible y funcionando 24 horas al día. Además de esto, hay una serie de requisitos secundarios relacionados con el coste de la infraestructura, y otros requisitos menos relevantes aún como pueda ser infraestructura adicional, tales como balanceadores de carga para la web o redundancia en los servidores para mejorar el tiempo en línea.

Se empezó haciendo un despliegue del *backend* de la aplicación (el servidor de Express, la base de datos, y los *scripts* de recogida de datos) en Amazon Web Services.

Amazon Web Services (o AWS) es una compañía subsidiaria de Amazon que ofrece servicios de computación en la nube bajo demanda. Lo usan empresas como Dropbox o Foursquare. Otras alternativas a este servicio podrían ser Google Cloud, Microsoft Azure, o para los servicios concretos que se iban a desplegar en este caso también Heroku o Now.sh.

En general cualquiera de estas compañías ofrecen lo que se buscaba para este proyecto: una plataforma de computación bajo demanda en la que desplegar nuestra aplicación en módulos

que se comunican entre sí, a coste muy bajo (con instancias sencillas), y de forma pública (en el caso de la web, esto es un requisito básico para poder acceder a ella).

AWS dispone de plantillas para despliegue de aplicaciones basadas en mongoDB, así como grupos de instancias para computación bajo demanda (AWS EC2). Desafortunadamente en este caso, estas plantillas suelen estar pensadas para empresas con expectativas de crecimiento y escalabilidad mucho mayores, por lo que el número de recursos que se despliegan (y el coste asociado) es mucho mayor.

Al final se optó por no usar estas plantillas y desplegar una instancia simple cubierta por el crédito básico que ofrece la plataforma para estudiantes (de tal manera que el coste de los recursos consumidos es gratuito). Con esta instancia cubríamos el requisito del tiempo en línea, pero sin embargo sí llevaba un pequeño coste asociado (por el tiempo de cómputo que no cubrían los créditos).

Como se desconocía el volumen final de la base de datos (que conllevaba un coste de recursos de almacenamiento), el posible volumen de tráfico de la app (que conllevaba un coste de ancho de banda y consultas a la web y al servidor), y siempre había un pequeño coste marginal mensual, se optó por no usar esta plataforma y pasar a usar un servidor casero con despliegue manual.

Este servidor es un ordenador con un Intel Core i3 a 3.7GHz, 8GB de RAM a 1600MHz, un SSD M.2 de 256GB y una interfaz de red Gigabit. El sistema operativo utilizado es Ubuntu 18.04 *desktop*. Se planteó utilizar la versión de servidor de este sistema operativo (sin entorno gráfico, con menor carga sobre el sistema), pero se descartó para poder usar la plataforma de manera más cómoda (para trabajar directamente sobre el código o consultar la base de datos mediante interfaz gráfica).

Para la web se aprovechó que el repositorio estaba *hosteado* en la plataforma de desarrollo Github (repositorios bajo *git*), ya que esta plataforma ofrece un servicio de *hosting online* para webs en cada repositorio, llamado Github Pages. Esto permite crear una web (normalmente de documentación o *landing* de proyecto) para cada repositorio del usuario. Gracias a esto, se pudo asociar una URL de Github pages a la web de la interfaz gráfica del proyecto.

3.3.3. Diseño de componentes: interfaces

En esta sección se detalla el esquema de diseño que se ha seguido durante el proceso de implementación a nivel de archivos del proyecto, denominados módulos (por la arquitectura del entorno).

En este apartado, cuando se hace referencia a la palabra interfaz, se refiere normalmente al conjunto de métodos que tiene un objeto para poder trabajar con este (concepto común en programación orientada a objetos). Cuando se hace referencia al cliente web se hablará de interfaz gráfica o interfaz de usuario.

Para ello, se describen en detalle las interfaces de entrada y salida y los componentes de todos los módulos implementados en las distintas partes del proyecto, así como los métodos y el flujo de cada uno de dichos componentes.

Los diagramas de clases representados a continuación tienen la siguiente estructura:

1. Nombre del archivo/clase
2. Dependencias del archivo (incluyendo submódulos importados)
3. Métodos implementados en dicho módulo.

3.3.3.1. Interfaces: Crawler El módulo *`crawler`*, como se ha descrito anteriormente, está compuesto de un módulo principal que ejerce la función de planificador y que invoca al resto de módulos importados para aglutinar todos los datos recogidos.

A continuación se describe cada uno de los módulos, su funcionalidad a nivel de interfaz y el detalle de los métodos implementados.

1. **Traffic**: el módulo `Traffic` importa las dependencias `http` (necesaria para hacer peticiones HTTP), `fs` (necesaria para leer y escribir archivos del sistema) y `parseString` (de la librería xlm2js, para *parsear* archivos XML).

   Este módulo implementa los métodos `traffic`, `getNN` y `getDate`. A continuación se

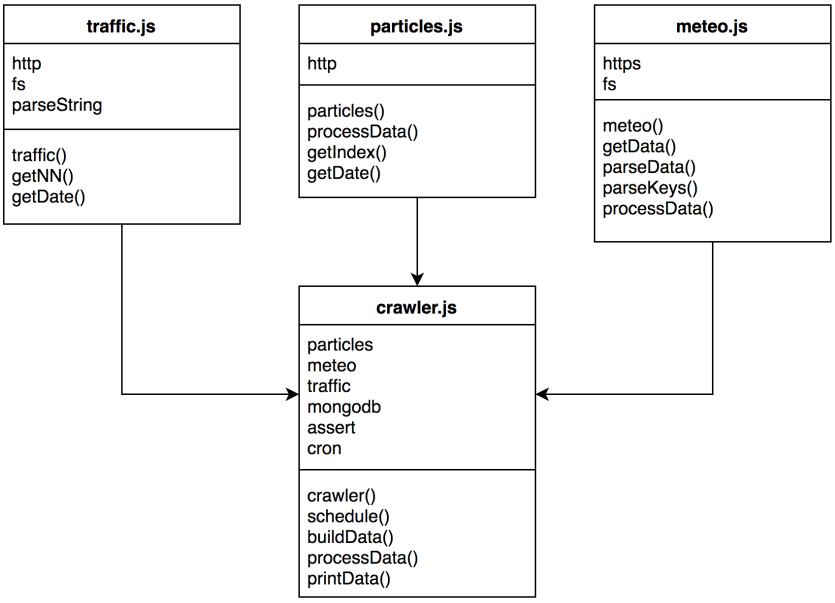

Figura 17: Interfaces del módulo *crawler*

detalla el flujo completo de este módulo en términos de funcionalidad e interfaz de entrada y salida.

El módulo `Traffic` recibe los datos en formato csv almacenados en la web de *munimadrid*, que se corresponde con la plataforma de datos abiertos del Ayuntamiento de Madrid. Por otra parte, una vez que procesa los datos, este método devuelve los datos en formato de objeto a través de un *callback*.

Esta forma de devolver los datos es típica de entornos y lenguajes de programación que funcionan de forma asíncrona. La función *callback* es una función que se pasa por parámetros al llamar al método principal, y que se llama desde éste con el objeto que se devuelve. Así, es posible trabajar con datos que pueden no estar disponibles inmediatamente (como, por ejemplo, los datos que se obtienen de internet) sin detener el flujo del resto de la aplicación (que puede no funcionar en paralelo). Estos datos asíncronos están disponibles en forma de *promesa*, que al cumplirse (cuando se reciben los datos), permite llamar a la función *callback*, devolviendo así los datos.

El flujo del submódulo es el siguiente:

a) Desde `crawler.js` se llama a la función traffic del módulo con el mismo nombre. Esta función lee un archivo de coordenadas ya guardado en el sistema, que se corresponde con los datos de las 24 estaciones de medición de calidad del aire con las que se trabaja en este proyecto. Entre otros datos, se pueden encontrar las coordenadas geográficas de cada estación.

b) A continuación, y con los datos que se obtienen del archivo, se realiza una petición HTTP al servidor donde se almacenan los datos públicos con el estado del tráfico. Estos datos están disponibles en formato XML, por lo que se *parsean* a tipo `Object` de Javascript para poder operar con ellos.

c) A continuación, se recorre el conjunto total de datos obtenidos extrayendo la siguiente información de cada punto de medición: coordenadas, intensidad, intensidad máxima (o de saturación), e ID del punto. Además de esto, se llama al método `getNN` que devolverá la zona más próxima (punto de medición de la calidad del aire) al punto de medición del tráfico que se está evaluando.

d) Con la lista de puntos de medición del tráfico mapeada o proyectada sobre la lista de zonas disponibles, se realiza la media de los valores (normalizados sobre el valor de saturación de la estación para obtener un valor porcentual entre 0 y 100). Esto permite obtener, para cada estación, un valor medio sobre 100 de la intensidad del tráfico para la estación dada. También se utiliza el método `getDate` para obtener la fecha a la que se corresponden los datos, en un formato específico que será útil a la hora de insertar y recuperar estos datos de la base de datos.

e) Por último, como ya se ha explicado antes, el objeto final con las 24 zonas con su correspondiente intensidad del tráfico y hora de los datos se devuelve a través del *callback* que se pasa por parámetros.

2. **Particles**: el módulo `Particles` importa sólo la librería `http` como dependencia. Implementa los métodos `particles`, `processData`, `getIndex` y `getDate`. A continuación se detalla el flujo completo de este módulo en términos de funcionalidad e interfaz de entrada y salida.

a) El módulo hace una petición http al servidor de datos abiertos correspondiente, donde recibe un CSV que almacena como lista de filas.

b) Esta lista de filas se pasa como argumento al método `processData`, el cual usa una lista de zonas previamente disponible para crear una lista de 24 objetos vacíos que serán los que se llenen con los datos correspondientes a las magnitudes medidas en cada estación.

c) A continuación, se realiza un pre-procesado de los datos eliminando los atributos innecesarios para el propósito del proyecto. Se filtran las columnas no numéricas (el csv tiene columnas intercaladas con valores V/F en función de si el dato está verificado o no, en valores no medidos el valor de la columna es "F" mientras que en valores medidos el valor es "V"), así como las primeras columnas correspondientes a IDs de municipio y región.

d) Se van rellenando los objetos en el diccionario de valores mapeando el ID de cada partícula en la fila dada, con el nombre de dicha partícula que ya está previamente almacenado. Se obtiene un objeto con clave-valor correspondiente al código numérico de la zona y objeto con las magnitudes medidas y su valor, una medición para las últimas 24 horas. Esto se puede traducir a una matriz de datos, donde hay 24 tablas (zonas) de 24 filas (24 horas), de X atributos, donde X es el número de magnitudes medidas en la zona dada.

e) Usando el método `getDate` se hace un cálculo con la hora a la que se corresponde la última medición no-nula, y llama al método `getIndex`, que devuelve el Índice de Calidad del Aire para la zona dada, así como la partícula más contaminante.

f) El método `getIndex` sigue el estándar de medición CAQI/ICA (o *Common Air Quality Index*). Según este estándar, el índice se corresponde con la cantidad medida de la partícula más contaminante, normalizada en una escala de 0 a 100 (aunque puede superar los 100 si la medición de dicha partícula supera el umbral más alto). Para ello se normalizan todas las partículas relevantes para el cálculo de este índice, si las hubiera en la medición de la estación dada. Después, se comparan los valores normalizados y se devuelve el que corresponda.

g) Una vez que se tiene el objeto de 24 zonas, cada una de las cuales tiene las medidas de magnitudes correspondientes, el Índice de Calidad del Aire, la partícula más contaminante, y la fecha de los datos, se devuelve este objeto usando el *callback* pasado por parámetros, como se ha explicado en el apartado anterior.

3. **Meteo**: el módulo `Meteo` importa las dependencias `https` (necesaria para hacer peticiones HTTPS) y `fs` (necesaria para leer y escribir archivos del sistema). A su vez, implementa los métodos `meteo`, `getData`, `parseData`, `parseKeys` y `processData`.

   El flujo de este submódulo es el siguiente:

   a) El método `meteo` ya tiene una lista de las estaciones de meteorología disponibles de las que se extraen los datos presentes en la documentación de los datos disponibles. A partir de esta lista de estaciones, se lee el archivo con las 24 zonas de Madrid, el cual detalla qué estación meteorológica corresponde a cada zona. A continuación, para cada una de las cuatro estaciones sobre las que se está trabajando, se llama al método `getData`, con un *callback* que toma los datos que devuelve este método, evalúa si se tienen todos los datos correspondientes a las cuatro zonas, y llama a `processData` si ese es el caso.

   b) El método `getData` hace una petición HTTPS a la API de datos abiertos de la Agencia Estatal de Meteorología, con la estación de la que se quiere obtener la observación correspondiente a las últimas 24 horas. Esta petición devuelve un objeto JSON con una nueva URL, a la que se hace otra petición para obtener, ahora sí, los datos meteorológicos de la estación especificada.

   c) Con cada uno de los objetos correspondientes a los datos de medición de la estación dada, se llama al método `parseData` para dar el formato adecuado a los datos. El método `parseData` hace uso del método `parseKeys` para transformar las etiquetas de las variables tales como "ta" (temperatura), "pres " (presión) a nombres más legibles (en caso de querer una salida más *verbose*). Este método también devuelve la lista de valores de forma más limpia para su procesado final, por lo que al final, para cada estación, se cuenta con una tabla de las últimas 24 mediciones y las magnitudes medidas en cada una.

d) Una vez se tienen todos los datos, el método `processData` transforma la lista de valores para cada estación en un objeto que se corresponde con la última medición, y que contiene tuplas clave-valor con el identificador de la magnitud medida y su valor. Al final habrá 24 objetos con esta estructura. Este formato de salida es el mismo que en los otros submódulos, y está estandarizado para facilitar el tratamiento de los datos en el módulo de *crawler*.

e) Una vez que se obtiene este objeto que contiene las 24 zonas, y para cada una un objeto con los valores medidos y la hora de la medición, se devuelve a través del *callback*.

4. **Crawler**: el módulo `Crawler` importa todos los submódulos anteriormente descritos, además de las dependencias `mongodb` (necesaria para realizar las operaciones *CRUD* sobre la base de datos), `assert` (necesaria para testear el entorno antes de la ejecución del programa) y `cron` (necesaria para planificar la ejecución del servicio una vez cada hora, similar al propio `cron` de BSD).

   A su vez, este módulo implementa los métodos `crawler`, `schedule`, `buildData`, `processData` y `printData`.

   El flujo de este submódulo es el siguiente:

   a) Lo primero que se hace tras comprobar que el entorno de desarrollo es el correcto (es decir, que la URL de la base de datos está bien definida) es llamar al método `scheduler`, el cual planifica un trabajo que llamará al método `crawler` una vez cada hora.

   b) A su vez el método `crawler` lo que hace es llamar a los submódulos descritos anteriormente en orden, con un *callback* en el cual se llama a la función `buildData` para ir construyendo los datos aglutinados, y al siguiente submódulo. En el último *callback* se llama directamente a la función `processData` tras el último `buildData`.

   c) En el método `buildData` simplemente se va poblando un diccionario con la tupla clave-valor tipo de datos (partículas, meteo o tráfico) y objeto con dichos datos (ver interfaz de los submódulos detallados anteriormente).

d) El método `processData` se conecta con la base de datos y comprueba que no haya errores. A continuación, y con los datos globales de los tres submódulos empleados, recorre las 24 zonas disponibles, inserta el valor del tráfico medido en la hora que da este submódulo (siempre más actual que los otros dos submódulos), y actualiza las filas correspondientes a los otros dos submódulos (meteo y partículas), realizando una búsqueda en la base de datos de la fila cuya hora (de tráfico) se corresponda con la hora de los datos medidos. Una vez hechas las operaciones de inserción y actualización sobre la base de datos, se cierra la conexión y se espera a que el planificador vuelva a llamar al método principal al cabo de una hora.

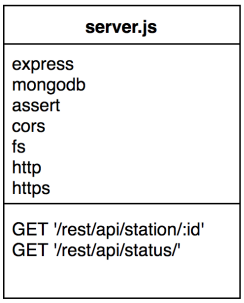

Figura 18: Interfaces del módulo de servidor

3.3.3.2. Interfaces: Servidor El módulo `Server` (o Servidor), como se ha descrito antes, está compuesto de un solo módulo principal que realiza las tareas de servidor HTTP(S) a través de una serie de librerías como `Express` detalladas anteriormente en la sección de Entorno y tecnologías utilizadas.

Este módulo importa las librerías de `express` (librería para servir rutas de acceso al estilo REST), `mongodb` (librería para acceder a la base de datos), `assert`, `cors` (librería para desarrollo que evita problemas de *CORS* que surgen al hacer peticiones a la misma máquina en distintos puertos), `fs`, `http` y `https` (que en este caso sirven para dar soporte a la escucha de peticiones a través de HTTP y HTTPS).

Este módulo no implementa ningún método, pero en el diagrama se han incluido las dos rutas sobre las que escucha peticiones para servir datos, las cuales se corresponden con una

consulta de las últimas 24 o 168 filas (una semana de datos) de la base de datos sobre una zona concreta, o con la consulta de la última fila para todas las tablas disponibles (en este caso, las 24 zonas del proyecto). Para simplificar, esta última consulta sólo devuelve datos concretos que se mostrarán en el componente Panel de la interfaz gráfica (detallado más adelante).

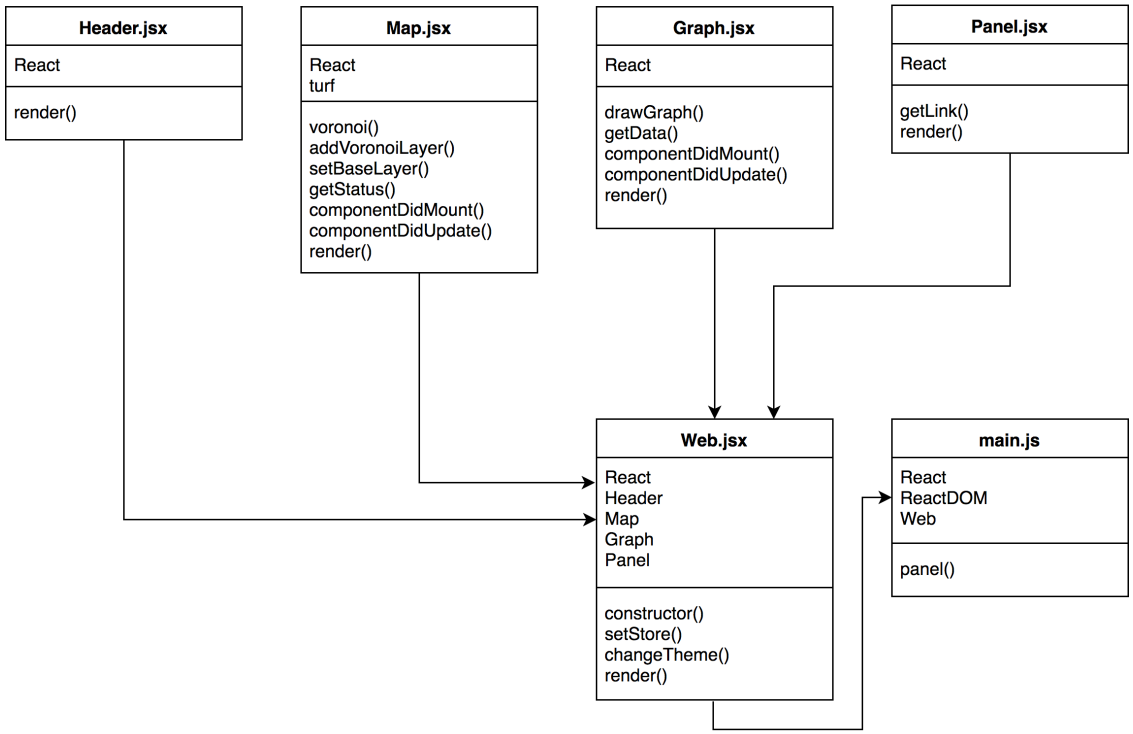

Figura 19: Interfaces del módulo de la interfaz de usuario

3.3.3.3. Interfaces: Interfaz de usuario (web) El módulo `Webui` (web UI, o interfaz web), como se ha descrito antes, se corresponde con el proyecto completo de interfaz web del proyecto. Esta parte del proyecto tiene su propio esquema de organización, así como dependencias independientes asociadas.

Esta sección del proyecto, como ya se detalló en el apartado de arquitectura y tecnologías utilizadas, está realizada empleando varias tecnologías de reciente desarrollo, como pueden ser librerías del tipo de React, *node modules*, *webpack*, *linter*, *babel*, etc. Una breve descripción de las más relevantes puede encontrarse en el apartado de tecnologías utilizadas.

Para la arquitectura de esta sección, es importante destacar todas estas librerías de desarrollo como *webpack* o *babel* que se encargan de transpilar código en estándares recientes a versiones más universales que tienen mayor tasa de adopción en dispositivos antiguos. Por lo tanto, la base del proyecto está montada alrededor de varias librerías que hacen esto posible.

1. En primer lugar, como proyecto de Nodejs (módulo de *npm*), se utiliza un archivo `package.json` que especifica las dependencias del proyecto, así como diversos scripts que se ejecutarán (normalmente usando diferentes `node_modules`) en diferentes situaciones.

2. Después utilizamos `webpack`, que es el empaquetador del proyecto. Este se encarga de transpilar el código Javascript ES6 a versiones antiguas, empaquetarlo, *minificarlo* (reduciendo espacios, líneas, y nombres de las variables si fuera posible, dificultando su lectura pero mejorando el tamaño final del archivo), y dejarlo listo para producción. A través del archivo `webpack.config.js` se especifican los parámetros que se usarán, tales como el directorio de salida y los diversos módulos que se obtendrán (se puede decidir empaquetar por separado librerías grandes y el código), así como optimizaciones, y diferentes librerías de carga para distintos formatos de archivo que se utilicen (por ejemplo, se puede decidir usar una librería para cargar especificaciones no tan extendidas de CSS como SASS).

3. En el caso concreto de este proyecto se utilizarán archivos *.jsx* para React, por lo que también se usará Babel como transpilador para los archivos que utilicen esta extensión. La configuración de Babel se encuentra en el archivo *.babelrc*.

4. Por último, en lo que al entorno de desarrollo se refiere, también existen archivos de estilo de editor y de *linter* (*eslint* en el caso de Javascript) que permiten seguir estándares de limpieza de código para facilitar la labor de mantenimiento futura.

Dentro del módulo, como se puede ver en el diagrama de clases, existe un archivo `main.js` que será el punto de entrada de la aplicación. Este archivo importa las librerías React y ReactDOM para leer componentes y trabajar con ellos en el DOM, respectivamente. Desde

este punto de entrada se carga el componente `Web`, que será el componente base o raíz de la interfaz.

1. **Web**: el componente `Web` importa los demás componentes de la interfaz, véase `Header`, `Map`, `Graph` y `Panel`. Además, este componente, al ser el componente raíz de la aplicación, implementa los métodos `constructor` y `setStore` (modificador del estado) que se pasarán por parámetros (o *props*) al resto de componentes hijos que lo necesiten. Además de esto, implementa un método `changeTheme` que cambia el tema de la interfaz entre tema claro y tema oscuro, y el método `render` estándar en todos los componentes de React y que sirve para renderizar la vista correspondiente.

2. **Header**: la única función de este componente, que es un componente "presentacional" (*presentational component*, esto es, no tiene lógica embebida, sólo carga componentes visuales), es renderizar la cabecera de la web con el logo en formato *.svg*. Como tal, sólo implementa el método `render`.

3. **Map**: este componente se encarga de toda la lógica relativa al mapa de voronoi que se muestra en el panel de la interfaz. Además de React, importa también *turf*, la librería de análisis de datos geo-localizados a la que se hace referencia en el apartado de tecnologías utilizadas. Este componente implementa los siguientes métodos:

    a) `setBaseLayer`: este componente se encarga de cargar una capa base de mapa sobre el mapa de *leaflet*, nuestra librería global para la creación del mapa que usaremos.

    b) `voronoi`: este método se encarga de calcular y representar las regiones de voronoi en el mapa. Con la ayuda del archivo con las coordenadas de los puntos de medición de la calidad del aire, crea una colección de *features* (término empleado en contextos de programación con datos geolocalizados, hace referencia a objetos con dichos datos) los cuales incluyen las coordenadas de la estación en cuestión, el nombre y el ID de la estación, y un color que representa el Índice de Calidad del Aire en dicha estación, obtenido en el método `getStatus` (detallado más

adelante). A partir de esta colección de *features*, obtenemos unos polígonos que representaremos en una capa del mapa superpuesta a la capa de callejero y que será nuestro mapa de voronoi con el particionado en base a los puntos anteriormente mencionados.

c) `addVoronoiLayer`: este método se encarga de añadir los polígonos anteriormente mencionados a una capa de `leaflet`. Además de añadir la capa, `Leaflet` nos permite añadirle interacción como *listeners* para eventos (por ejemplo, al hacer click en un polígono, esto es, una región del mapa). Esto nos permite mostrar información adicional sobre la zona que seleccionemos.

d) `getStatus`: este método hace una llamada al servidor HTTP de express de nuestra aplicación, obteniendo los datos que necesitamos referentes a la ruta `status`, es decir, el estado global de la calidad del aire y otras variables (ver sección del módulo `server`). Recibe los datos y los pasa a través de una función *callback* que en este caso se encuentra en el método `voronoi` y es la función *lambda* encargada de continuar con el proceso de renderizado del mapa de voronoi.

e) `componentDidMount`: este método es estándar de React, y se le llama cuando la estructura básica de la vista termina de cargarse. Esto nos permite cargar el mapa, cargar la capa básica del mapa con `setBaseLayer`, y el mapa de voronoi con `voronoi` una vez que ya han cargado los elementos del DOM donde vamos a acoplar estos componentes.

f) `componentDidUpdate`: este método también es estándar de React, y se llama cada vez que los *props* (o parámetros) de un componente cambian. En nuestro caso, este método se llama cuando cambiamos el tema de la interfaz desde el componente padre `Web`. Este método elimina la capa base, carga una nueva capa base con el color correspondiente, elimina la capa de voronoi, y recarga la capa de voronoi sin regenerar los polígonos ya calculados, sólo para cambiar los colores si fuese necesario.

g) `render`: por último, tenemos el método `render`, que carga un simple *div* sobre el que acoplaremos el mapa y sus correspondientes capas.

4. **Graph**: este componente no importa ningún módulo además de React, aunque sí hace uso de la librería global D3 para crear visualizaciones de datos. En nuestro caso, además de los métodos `componentDidMount` (que llama a `drawGraph` para representar el gráfico con los datos en forma de serie temporal), `componentDidUpdate` (que elimina el gráfico y lo vuelve a pintar con información nueva), y `render` (que carga un contenedor para el gráfico), tenemos los métodos nuevos:

   a) `getData`, que hace una petición al servidor de la aplicación para obtener datos en histórico de la zona seleccionada

   b) `drawGraph`, que hace uso de la librería D3 para pintar las gráficas correspondientes a las variables de interés en la zona seleccionada.

5. **Panel**: por último, este componente se encarga de representar unas tarjetas informativas para la zona seleccionada, cada una con el atributo o variable de interés y un código de color en función de la gravedad del valor representado (saturación del tráfico, calidad del aire, etc.). Estas tarjetas usan los datos obtenidos por el componente `Map`, que se cargan en el estado del componente padre `Web` al hacer click en una de las zonas, y se propagan a este componente mediante los *props* anteriormente mencionados.

3.3.4. Herramientas de desarrollo

Este apartado detalla las herramientas utilizadas durante todo el proceso de desarrollo, así como otras herramientas complementarias que han sido útiles durante dicho proceso.

1. **Visual Studio Code** es un editor de código gratuito optimizado para el desarrollo de aplicaciones web, creado por Microsoft. A pesar de estar desarrollado con el *framework* Electron para la creación de apps universales con tecnologías web (famoso por su bajo rendimiento en comparación con otros editores ya que necesita una instancia de *Chromium* y Nodejs para funcionar), su rendimiento ha mejorado mucho recientemente. Las herramientas que proporciona este editor para el desarrollo de aplicaciones web, así como diversos paquetes que facilitan el proceso de desarrollo han sido de gran utilidad a lo largo de todo el ciclo de desarrollo del proyecto.

Los paquetes de Visual Studio Code (o VSCode) con los que se ha trabajado para este proyecto son: `Color Highlight` (para el resaltado de las paletas de color usadas en la visualización de datos del mapa y el gráfico, así como en las hojas de estilos), `ES7 React/Redux[...] snippets` (para autocompletado de funciones y estándares de dichas librerías) y `ESLint` (para unificar el formato del código producido).

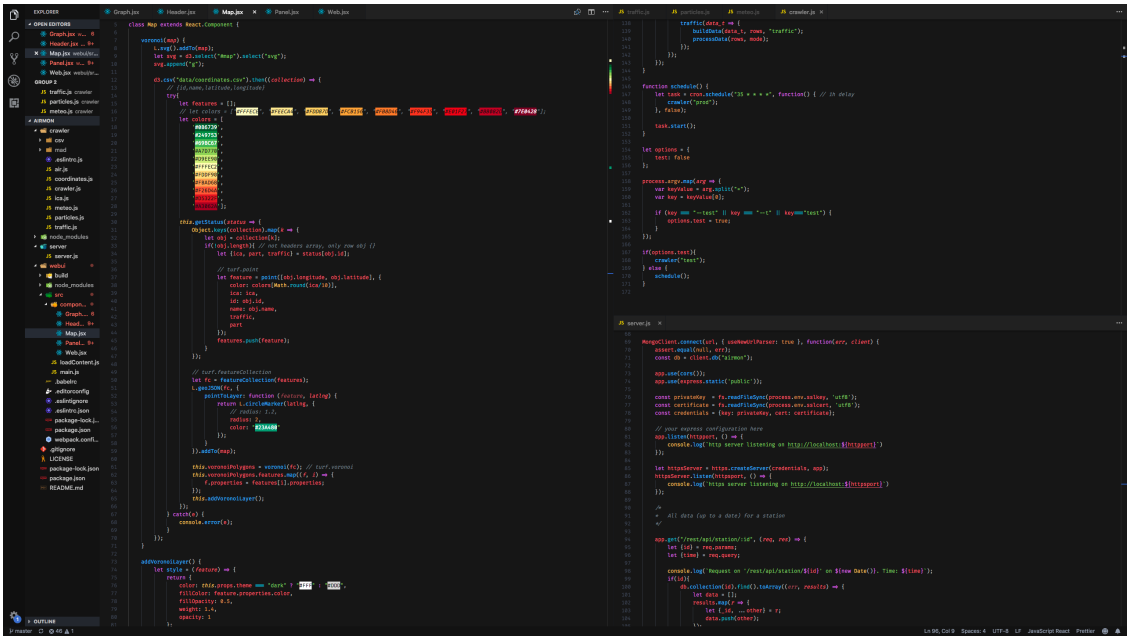

Figura 20: Visual Studio Code en la implementación del proyecto

2. **Mozilla Firefox Developer Edition** y las herramientas de desarrollo del propio navegador. Este navegador, en su versión 63 de desarrollo, hace uso de la tecnología *Quantum* de Mozilla, que promete un alto rendimiento así como una gran cantidad de características embebidas en el navegador de forma nativa. De cara a las herramientas de desarrollo web de Firefox (o *DevTools*), tenemos un inspector de DOM mejorado, una consola con bastante flexibilidad en salida e interacción (permite por ejemplo plegar o desplegar campos de objetos en la salida por consola), un *debugger* integrado con el código Javascript de la página, un monitor de red con opciones de *throttling* o monitorización de las distintas peticiones de red a lo largo del tiempo, un panel de datos almacenados por la web, o un modo *responsive* para probar diversos tamaños o dispositivos sobre la web, entre otros.

Además de esto, se ha usado una extensión React Developer Tools, que nos permite no sólo interactuar con los componentes en el DOM sino también ver los *props* en cada componente y entender mejor el árbol de componentes.

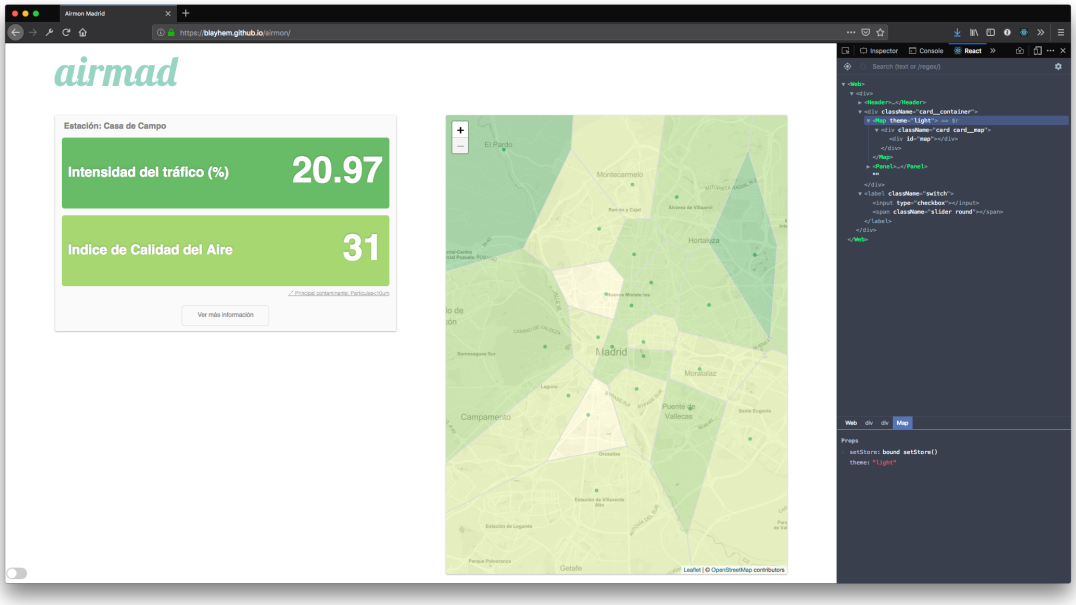

Figura 21: Una versión en desarrollo de la web en Firefox Dev Edition con React Dev Tools

3. **MongoDB Compass**: esta herramienta es una interfaz gráfica para la base de datos de mongoDB. Permite explorar los datos de manera visual, hacer búsquedas y realizar operaciones *CRUD* directamente desde la interfaz. Gracias a esta herramienta podemos revisar que el *crawler* va guardando los datos en las tablas correspondientes, así como consultar posibles filas incompletas o inexistentes con el fin de *debuggear* el código correspondiente.

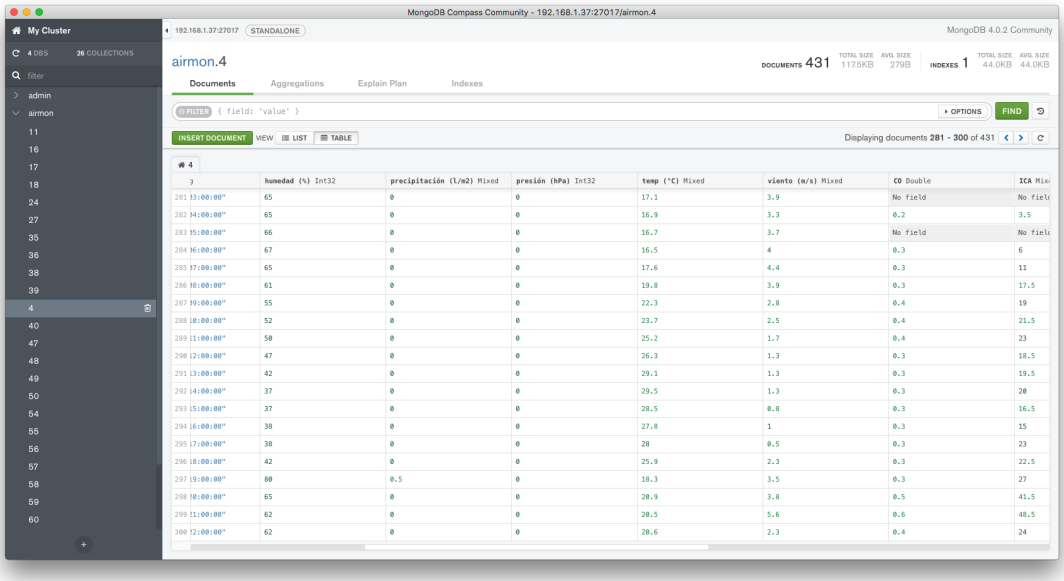

Figura 22: La interfaz de MongoDB Compass en la tabla correspondiente a la zona 4

3.4. Implementación

Esta subsección trata de manera superficial el proceso de desarrollo e implementación del proyecto. Puesto que ya se ha desarrollado previamente la arquitectura del proyecto en detalle, en este apartado se quieren aclarar procesos o flujos del proceso de desarrollo implícitos en esta fase.

En primer lugar, y puesto que era la base de la interfaz visual que presentaría el proyecto, se desarrolló toda la capa de recogida y almacenamiento de datos: los *scripts* correspondientes al módulo Crawler y la base de datos. Para ello, y como se mencionó anteriormente, se definió una interfaz común de salida para los diferentes módulos de tal modo que fuese sencillo adaptarlos e integrarlos en la aplicación principal.

Dentro de esta parte, el mayor reto encontrado fue la conversión de coordenadas entre las distintas fuentes de información. Las coordenadas correspondientes a los puntos de medición del aire se encontraban en el estándar *EPSG:25830 ETRS89 / UTM zone 30N*, de uso típico en la Unión Europea, mientras que los puntos de medición del tráfico y algunos datos de referencia venían dados en el estándar *EPSG:4326 WGS 84*, usado típicamente por sistemas

GPS o software como Google Maps, a nivel global. Estos sistemas no tienen una conversión sencilla ya que el marco de referencia (Europa) va cambiando su posición ligeramente anualmente. Finalmente se encontró una API donde se realizaba esta conversión, y se pudo desarrollar un módulo adicional, `coordinates.js`, del cual se pudo conseguir las coordenadas de los puntos de medición de la calidad del aire, en el mismo formato que en el resto de fuentes. Esto era necesario para la fase de clasificación de los datos y clusterización de las estaciones de medición (de tráfico y meteorología) en las 24 zonas correspondientes.

Otro problema relativo a la base de datos fue encontrar un estándar común de fecha y hora para actualizar los datos, ya que las búsquedas para inserción son sensibles a ceros en los dígitos. Debido a esto, datos como 08/09/2018 y 8/9/2018 no quedaban relacionados, no pudiendo actualizar filas de la base de datos, y perdiendo información. Para solucionarlo, se unificaron todas las fechas y horas de los datos antes de su procesado añadiendo ceros en números de un dígito.

A continuación, y partiendo de la base de los diseños realizados sobre los requisitos planteados, se pasó a realizar un prototipo de la interfaz web. Como ya se mencionó al hablar del entorno, al ser un proyecto de React era necesario configurar muy bien todas las librerías que rodeaban el proyecto, como Webpack, Babel, etc. Se partió de la base de un proyecto antiguo con una estructura similar, se revisaron todos los archivos de configuración y se montó la estructura base del nuevo proyecto.

A partir de ahí, se fueron creando los distintos componentes de la interfaz, añadiendo funcionalidades progresivamente y modularizando más en función de la complejidad de los componentes.

Uno de los retos encontrados en esta sección ocurrió al querer trazar el mapa de voronoi sobre el mapa base en leaflet. Los tutoriales y documentación encontrados para esto usaban D3, y dado que era la librería utilizada en el trazado de los gráficos de series temporales, se intentó reutilizar para el mapa para minimizar las dependencias del proyecto. Sin embargo, esto no fue posible, y se acabó utilizando turf (turf.js) para el tratamiento de los datos y generación de puntos y polígonos sobre el mapa. Turf era más sencilla de usar, y debido a las limitaciones temporales del proyecto, se consideró la mejor opción.

4. Evaluación

En esta sección del documento se describe el proceso de evaluación llevado a cabo para probar la funcionalidad del proyecto desde el lado del usuario.

Para probar la funcionalidad completa del proyecto se planteó un conjunto de pruebas sobre el código usando diversas librerías para pruebas unitarias y de integración y para pruebas de interfaz. Para evaluar la usabilidad de la interfaz se han utilizado tests de guerrilla sobre diversos perfiles de usuarios para evaluar posibles fallos de diseño.

4.1. Entorno de pruebas

Tests de funcionalidad: Antes de hablar del conjunto de pruebas establecido para esta sección, conviene hacer una introducción a los tests unitarios, funcionales y de integración.

Los tests unitarios prueban pequeñas funcionalidades del código, a menudo funciones, de forma aislada. Si el test usa recursos externos (conexiones a base de datos o a otras librerías) no es un test unitario. En teoría, los tests unitarios deben ser fáciles de implementar, ya que sólo debe de probar un conjunto de posibles entradas y esperar la salida deseada.

Los tests de integración prueban la integración de diversas partes del proyecto. Se diferencian con respecto a los tests unitarios en que estas pruebas no son aisladas. Se usa en aquellas situaciones en las que un test unitario no es suficiente (por ejemplo, para probar la integración de un módulo con la base de datos). Son más complejos que los tests unitarios, y por lo tanto se priorizan estos últimos, a menos que sea necesario implementar tests de integración.

Los tests funcionales comprueban la funcionalidad completa de la aplicación. En aplicaciones web esto requiere automatización de uso del navegador. Estos tests son mucho más complejos de implementar y mantener que los anteriores [45].

Las librerías para pruebas disponibles para código en Javascript se pueden dividir según varios criterios: o bien por el tipo de test en el que se aplican, o bien por la funcionalidad que aportan para las pruebas (estructura, afirmaciones -*assertions*- pruebas con datos falsos,

cobertura o librerías para navegador) [46].

En este documento no se entra en detalle en esta segunda organización ya que en este caso la categorización que se ha seguido ha sido la definida por el tipo de test a aplicar en cada caso.

Para las pruebas unitarias, se siguen los siguientes criterios:

1. Se comprueba que la salida es siempre la deseada con herramientas de afirmación (o *assert*).
2. Se usan herramientas de cobertura para comprobar que todas las unidades mínimas o funciones están cubiertas.

Para los tests de integración, se usan datos falsos (y, en el caso concreto de nuestro entorno de pruebas, tablas de la base de datos específicas para pruebas).

Dado que la funcionalidad de la interfaz desarrollada es simple, no se llevan a cabo tests funcionales, sino que esto se prueba más adelante en los tests de usabilidad con usuarios.

Para tests unitarios se planteó usar la librería `Jest`, recomendada por Facebook. Esta librería es conocida por tener buenas métricas de rendimiento, una interfaz clara, módulos de cobertura de código, e incluir varios paquetes que cubren casi todas las posibilidades de tests a implementar. Sin embargo, y como veremos más adelante, hubo que adaptar el tipo de pruebas por la naturaleza del proyecto.

Para las pruebas de integración se estudiaron las librerías `Jest`, `Mocha` y `Supertest`.

Tests de usabilidad: Para probar la usabilidad de la interfaz web de cara al usuario, se han empleado unas pruebas llamadas "tests de guerrilla". Según Jakob Nielsen en [47], "los tests de usabilidad elaborados son una pérdida de tiempo. Los mejores resultados se obtienen al probar con no más de 5 usuarios y con la mayor cantidad de pruebas pequeñas que se puedan realizar".

Según este método, realizando las pruebas de usabilidad de la aplicación o interfaz con 5 usuarios distintos ya podemos detectar el 85 % de los errores de usabilidad.

Este método se lleva a cabo con una serie de entrevistas a dichos usuarios, divididas en tres fases: preguntas de calentamiento y perfilamiento, preguntas de impresión (sin usar el producto) y evaluación mediante tareas a realizar por el usuario.

Durante todo el proceso es importante que el usuario comparta su tren de pensamiento en voz alta, para que se tenga en cuenta las decisiones y la intuición del usuario a la hora de sacar conclusiones sobre el producto.

4.2. Descripción de las pruebas y resultados

Tests de funcionalidad: El principal problema encontrado a la hora de implementar los tests de funcionalidad sobre la aplicación es el siguiente: Si se estudian los diagramas de interfaz de los diversos componentes del proyecto, podemos ver que

1. La capa de servidor no implementa ningún método
2. La parte de la interfaz de usuario sí implementa varios, pero son:
    - Métodos propios de React
    - Métodos con dependencias externas
3. La base de datos no tiene sentido incluirla en este tipo de pruebas
4. Los métodos principales del módulo Crawler tienen dependencias externas.

Por lo tanto, es evidente que los tests unitarios quedan fuera del alcance de este proyecto para probar la funcionalidad de la aplicación. Como ya se ha comentado, la mejor alternativa con dependencias externas serían los tests de integración.

Los tests de integración que se llevan a cabo incluyen

1. Tests del módulo servidor, donde se prueban las dos rutas disponibles: `/rest/api/status` y `/rest/api/station/:id`. Estos tests comprueban la integración del servidor con la base de datos con las consultas correspondientes a las dos rutas ofrecidas.

2. Tests del módulo Crawler, probando la funcionalidad de cada uno de los módulos (considerando la conexión de estos módulos con la fuente de datos externa al proyecto).

Tests de usabilidad: A continuación se incluye una tabla con los resultados de las pruebas para cada uno de los usuarios y cada una de las preguntas y tareas realizadas.

¿Cuál es su nombre?	Cristina	Jacobo	Antonio	Pilar	Montse
¿A qué se dedica?	Gestora de proyectos	Estudiante	Profesor	Limpieza	Investigación (UX)
¿Navega habitualmente?, ¿Cuántas horas navega al día?	Más de 12	8	3-4	1	1-2
¿Qué dos sitios web suele visitar de manera recurrente?	Instagram y twitter	Twitter y YouTube	Gmail y prensa digital	Noticias	Facebook, Instagram, Amazon
Edad	24	21	55	54	43
¿De qué cree que trata el sitio web?	Aparcamientos, vivienda (en Madrid) más barata o más cara, densidad de población, zonas verdes...	Calidad del aire en diferentes zonas. Aire por el nombre, calidad por los colores.	Algo geográfico (por las áreas). Información de algún dato por áreas de Madrid (población, tráfico...)	Indicativo de más uso (de ¿?) por zonas	Pisos (alquiler, alquiler turístico...) por zonas. Relacionado con airbnb

¿Qué acciones piensa que se pueden realizar en él?	Ampliar el mapa (está dividido por zonas), ver información para cada zona...	Analizar qué zonas tienen peor calidad del aire	Consultar algo de información sobre la zona que se seleccione	Navegar por el mapa	Puedes seleccionar un área y verás resultados de algo
Impresiones generales de la interfaz	El logo debería de verse más	Muy simple (positivo), le gustaría que el mapa ocupase toda la mitad derecha.	Las líneas divisorias no se ven bien en algunos casos. Faltan estaciones de medición.	Las líneas no se ven bien (en color y en grosor)	Más explicativo, no sé qué significan los colores y la leyenda del panel no es suficiente
Tarea 1: ver el ICA de una zona	No se esperaba primero otros factores (primero ICA)	Le gustaría tener algo más de información (menos genérico) en el panel inicial.	¿El usuario no puede volver atrás? (Deseleccionar zona)	Esperaría ver el ICA directamente en el mapa	Le gustaría que el ICA tuviera una leyenda y que estuvieran explicados los puntos de medición. Lo buscaría haciendo hover o pinchando

Tarea 2: ver los datos relativos a una zona	No se puede ir para atrás (cerrar gráfico o panel). Le gusta encontrarse números grandes y con color.	Muy relacionado con la 1. Le gustaría que el panel ocupase más espacio vertical y se plegase al mostrar el gráfico	Hace falta explicación sobre los datos, y unidades de medida. Sería interesante hacer más grande el detalle de la partícula contaminante. Separar los tipos de datos (partículas, tráfico, meteorológicos...) con títulos o similar.	Echa en falta información en grande (tamaño de la fuente), y que el panel ocupase toda la mitad izquierda.	El dato de partícula más contaminante se ve pequeño. Le gustaría que las cards ocupasen toda la mitad

Tarea 3: ver el gráfico del histórico de datos	Esperaría encontrarse una lista de datos coloreados (como referencia mala). Cambiar literal + info por 24h o algo así. Etiquetaría los ejes con unidades (días/horas, porcentaje...)	Cambiaría el literal (más información) por algo menos genérico (visualizar gráfica o similar). Echa en falta una explicación sobre el ICA (valor inverso contra-intuitivo, se espera que 100 sea bueno y 0 malo).	Se ve a lo largo del día pero echa en falta días anteriores. Si estás en vista de semana, que se pueda pinchar en el día y ver la vista de 24h.	Esperaría encontrarse la gráfica desplegada nada más pinchar en una zona. Entiende las correlaciones entre las líneas.	Cambiaría literal de + info a "ver histórico.º similar. Falta info en los ejes de la gráfica

Tabla 15: Pruebas de usabilidad con usuarios - Preguntas y resultados

5. Gestión del trabajo

En este apartado del documento se detalla la planificación realizada para llevar a cabo todas las fases de diseño y desarrollo del proyecto. Además de esto, se presenta un cálculo del presupuesto total en base a los costes estimados del proyecto.

5.1. Planificación

En esta sección se explica la metodología seguida durante el proceso de desarrollo del proyecto, así como los hitos conseguidos a lo largo de la duración de éste.

La metodología de desarrollo que se ha seguido para este proyecto es una metodología ágil basada en Scrum [48]. Esta metodología, frente a otras metodologías como el modelo en cascada, permite empezar el proyecto con una fase de planificación muy básica seguida de una fase de implementación o desarrollo del producto mínima: el producto mínimo viable (o *M.V.P* en inglés). A continuación, esta versión mínima del proyecto se prueba y se lanza para a continuación iterar sobre ella de nuevo a la fase de planificación. Estas iteraciones se conocen como *sprints*, duran entre 1 y 3 semanas, y van definiendo el producto final progresivamente.

En esta metodología hacen falta 3 roles: el rol del encargado de producto, que define las ideas que se van a llevar a cabo, el rol del *scrum master*, que lidera y gestiona el equipo y el proceso, y el equipo, que está compuesto por los perfiles de diseño, desarrollo y pruebas que hagan falta para el proyecto.

En este caso, el rol del encargado de producto venía subordinado del objetivo del proyecto y de los requisitos obtenidos, y el rol del *scrum master* iba muy ligado al equipo ya que todos estos perfiles han sido cubiertos por la misma persona. Esto tiene ventajas e inconvenientes, siendo la ventaja principal el hecho de que las estimaciones fuesen mucho más precisas que en otras circunstancias, y la desventaja principal una excesiva diversificación de perfiles (necesaria, por otra parte).

La metodología final ha resultado ser una adaptación de este proceso ya que se han obvia-

do procesos y artefactos (documentos) necesarios para la comunicación en un equipo pero redundantes en este caso.

A continuación se detallan los objetivos o metas conseguidas en diferentes jornadas de trabajo a lo largo de la duración de este proyecto.

12-22/03/2018	Primera version del proyecto con scrapper de calidad del aire (por colores)
13/04/2018	CLI para la consulta de particulas (inicialmente air.js)
01/05/2018	Añadidos algunos archivos y referencias necesarios
18/05/2018	Datos meteorológicos añadidos (meteo.py)
19/05/2018	Desglose del scrapper por hora (importante para la DB)
20/05/2018	Datos de tráfico añadidos
28/05/2018	Arreglado el sistema de coordenadas (8d)
29/05/2018	1-NN clasificación de las estaciones de tráfico
04/06/2018	Notebook con los resultados de 1-nn, experimentos de stats
05/06/2018	Reestructurando scrapper + eslint
06/06/2018	Meteo.py es ahora meteo.js (todo el proyecto en node.js)
29/06/2018	Particles ahora imprime datos en tabla (preparación de los datos)
04/07/2018	Primer borrador de memoria
09/07/2018	Comienzo con una primera versión del crawler
11/07/2018	Plantilla de la interfaz web (archivos de config para reactjs)
22/07/2018	Reestructuración del output en particles, preparado para tabla
25/07/2018	Reestructuración de meteo para output en tabla
27/07/2018	Integración con el crawler a través de callbacks, definición de interfaz (métodos)
30/07/2018	Integración de particles con una primera versión de DB en mongo
31/07/2018	Integración de particles y meteo con la DB
01/08/2018	Integración de tráfico con el crawler
03/08/2018	Revisión de dependencias para la webUI (turf etc)
04/08/2018	WebUI - logo, mapa con leaflet, dependencias.
05/08/2018	Voronoi con d3 y luego turf en leaflet, tema oscuro

06/08/2018	Arreglado el mapa de voronoi con turf, registrado issue para colorear mapa
12/08/2018	Preparada la inserción de datos en la base de datos
14/08/2018	Integrada la actualización de datos en la base de datos, actualizando filas con valores horarios cambiados, preparando el servicio de cron para planificar automáticamente la recogida de datos
17/08/2018	Revisado el estilo de los módulos añadidos hasta el momento con la ayuda de ESLint
20/08/2018	Reformateo de las fechas con las que trabaja el módulo crawler para la BBDD
21/08/2018	Actualizado el README de la aplicación, creada la plantilla del servidor de express y realizada una auditoría de seguridad sobre las dependencias del proyecto (npm audit). Componente de mapa llevado a un componente a parte del componente Web, añadido componente Graph con datos falsos para pruebas. Lanzamiento de la web en Github Pages, actualización de dependencias y revisión de estilos con ESLint.
22/08/2018	Arreglado un error del tema oscuro en Safari, se añade interacción al mapa mediante métodos de escucha a eventos de click sobre las zonas. Se añade la paleta de color al mapa y se prepara el servidor de express con la integración con mongoDB y las consultas pertinentes a este servidor desde los módulos de la interfaz.
23/08/2018	Se obtiene un primer valor del ICA a través de la web de eltiempo.es y se integra el crawler con las zonas definitivas (tablas) de la base de datos.
24/08/2018	Se estandariza la salida del módulo de ICA, se arregla un fallo de salida en el módulo de partículas y finalmente se integra el ICA en el módulo de partículas, eliminando el primer módulo mencionado.
25/08/2018	Se normalizan los datos en el cálculo del ICA, se incluye en la salida la partícula más contaminante, y se arregla el formato de las salidas de los módulos de meteo y partículas.

26/08/2018	Se preparan las rutas del servidor y se integran del todo en el código de la interfaz web. Se estructura el contenido en la web usando CSS grid y se aplican los colores reales sobre el mapa.
28/08/2018	Se soluciona un error relacionado con el coloreado del mapa y se añade la escucha sobre HTTPS al servidor para la integración con Github Pages.
29/08/2018	Se añade el componente panel para mostrar información sobre las partículas en formato de texto, se actualiza la ruta de estación/id en el servidor para admitir diferentes rangos temporales, y se da valores reales a la UI del panel y del gráfico.
31/08/2018	Se solucionan pequeños errores en el servidor de express.
01/09/2018	Se solucionan errores relacionados con las fechas de los datos al actualizar la BBDD.
02/09/2018	Se solucionan más errores relativos a las fechas de los datos, se mejora la frecuencia de refresco de los datos meteorológicos y se le aplica estilo a las tarjetas del panel en base al diseño visual de referencia.
11/09/2018	Se arreglan los últimos errores relacionados con fechas, y se cambia el formato de impresión por pantalla del servidor para facilitar la trazabilidad de errores.

Tabla 16: Planificación del proyecto, desglose por días

5.2. Presupuesto

En esta sección se detallan los costes directos (que comprende los costes de personal y los costes de equipo) y los costes indirectos precisos para la realización de este proyecto.

5.2.1. Costes de personal

A continuación se detallan los gastos asociados al coste de los trabajadores del proyecto. Para la estimación del coste de personal se ha realizado una investigación a partir de encuestas

anónimas del sector [49][50] y valorando el nivel de experiencia del trabajador necesario en los distintos roles del proyecto.

Para obtener el coste total por este concepto, se ha realizado una estimación del número de horas con respecto a los hitos conseguidos en la realización del proyecto. Los sueldos tomados como referencia son:

1. Correspondiente a un desarrollador web *fullstack* en una empresa de producto, de 3 a 5 años de experiencia, que oscila entre 30k y 40k € brutos anuales.
2. Correspondiente a un diseñador junior (menos de 3 años de experiencia), que de media está en 20.5k € brutos anuales.

Para calcular el coste bruto por hora de cada perfil, tenemos en cuenta que en 2018 hay un total de 251 días laborables. Si dividimos estos sueldos entre 251 tenemos 140€ brutos por día en el caso del desarrollador y 82€ brutos por día en el caso del diseñador, es decir, 17.5€ brutos la hora del desarrollador y 10.25€ brutos la hora del diseñador.

Perfil	Coste/hora	Horas	Coste total (€)
Desarrollador	17,5	400	7000
Diseñador	10,25	24	246
Total			**7246**

Tabla 17: Costes de personal

5.2.2. Costes de equipamiento

Para esta sección se han tenido en cuenta una serie de consideraciones sobre el material utilizado durante la realización de este proyecto. Se han tenido en cuenta:

1. Como punto de partida, el coste de los equipos empleados en relación al tiempo de vida de los mismos con respecto a la duración del proyecto. En este caso, la duración del proyecto ha sido de 6 meses.

2. El coste de infraestructura contratada para el proyecto (e.g. instancias empleadas de Amazon Web Services).

3. Los costes de electricidad y otros costes derivados de infraestructura.

Equipo	Coste unitario (€)	Coste aplicable (€)
MacBook Pro 13"2017	1580	158
Servidor linux i3, 4GB, 256GB SSD	592	59
Amazon Web Services, varios	11,65	11,65
Recibo electricidad servidor	8,93	8,93
Total		**237,58**

Tabla 18: Costes de equipo

5.2.3. Costes indirectos

Para los costes indirectos del proyecto -tales como facturas de luz del equipo de desarrollo, agua, teléfono, acceso a internet y otros- se estima una tasa del 20 % sobre el total.

5.2.4. Coste total

En definitiva, el coste total estimado para la realización del proyecto quedaría de la siguiente manera:

Tipo de coste	Cantidad (€)
Costes de personal	7246
Costes de equipo	237,58
Total c. directos	7483,58
Costes indirectos	1496,72
Total	**8980,3**

Tabla 19: Costes totales del proyecto

6. Consideraciones y conclusiones

En esta sección se describen las conclusiones, objetivos y consideraciones que se han recopilado durante la realización de esta práctica, con el objetivo de analizar el trabajo realizado de cara a proyectos futuros.

6.1. Conclusiones del proyecto

Se han cumplido todos los objetivos que se marcaron durante las primeras fases del proyecto -análisis y definición de alcance- que servían de indicadores para determinar la completitud del trabajo realizado.

En resumen, se ha conseguido

1. Obtener un conjunto de scripts que recogen datos de fuentes públicas para usarlos más adelante.

2. Montar una infraestructura de base de datos automatizada con el script anteriormente dicho para realizar la recogida de datos de forma periódica y tener los datos disponibles en mayor número y en cualquier momento.

3. Crear un proyecto de interfaz web con React que muestre al usuario los datos anteriormente recogidos.

4. Usar metodologías de diseño y desarrollo de producto para definir los requisitos, diseñar la interfaz, desarrollar prototipos y evaluarlos con usuarios.

5. Desarrollar la aplicación en global integrando todos los módulos, con estilos de código globales, y correctamente empaquetada y publicada.

6. Escribir este documento que acompaña al trabajo realizado, que sirve como memoria del proceso de realización del proyecto y como documentación del mismo.

Además de esto, se encuentra el hecho de que este proyecto empezó siendo un proyecto personal. Como tal, se considera como extra la satisfacción de haber escalado dicho proyecto a uno de mayor envergadura.

6.2. Proyectos futuros

La evaluación del proyecto final con usuarios dio pie a una siguiente recogida de requisitos para una futura iteración del proyecto.

Además de esto, y como bien se ha detallado en la sección correspondiente a la arquitectura de la aplicación, durante todo el proceso de desarrollo de este proyecto se ha hecho un fuerte énfasis en la modularización de la aplicación de cara a su adaptación en futuras versiones del proyecto.

Cualquiera de las capas del proyecto es flexible en cuanto a su implementación, siempre y cuando siga su funcionalidad. Esto permite la inclusión de nuevos módulos o la adaptación de módulos existentes para añadir funcionalidad o actualizar fuentes de datos.

La interfaz web, por otra parte, es relativamente flexible al utilizar una arquitectura por componentes que modulariza las partes de la propia web.

Todo esto permite, como se ha comentado, reutilizar módulos o secciones de este proyecto en proyectos futuros de temática o funcionalidad similar.

7. Summary

7.1. Introduction

This document contains the specifications and requirements for the design, implementation and testing of a suite of visualizations over the quality of the air in the city of Madrid.

Context In this period of time, people are moving from rural to urban areas, which creates a larger concentration of vehicles. This causes an increment in the level of dangerous particles in the atmosphere, which may lead to potential health issues in the population.

The main objective behind this project is to provide a product that can be used to consult the status of the air quality in the area people work or live in (in Madrid). This project aims to facilitate the consultation of data relative to the pollution that affects the atmosphere and environment, and to help understand the correlations between different variables that may affect said index.

The target user of this project consists of two persona: the main basic user, which may want to go to the dashboard to look for information related to the status of the air quality, and maybe explore the other data quickly. The other target focuses on a more experienced user, who uses the tool wanting to explore the data more deeply, and understand which factors cause changes in this air quality index.

The objectives set for this project are:

1. To deal with and show the biggest amount of information available, in different degrees of complexity, using advanced data visualization techniques.

2. To follow a design process based on research, with the objective of defining perfectly the project and its development process.

3. To follow an open-source based development process, with quality standards and code formatting. Code is going to be publicly available.

4. To integrate and document every layer of the project, from back to front.

This project uses two APIs as source of the data: the AEMET (Meteorology State Agency) API and the Open Data API from Madrid City Council. Some insights about the legal aspects in the terms and conditions of this APIs are:

- Both concur that the responsibility of the use of the data depends on the user
- Both rely on attribution when reusing the data (which does not apply in this project)
- AEMET specifically prohibits any illegal activity using the data.

7.2. State of the art

This section presents the reader with information relative to the background of the project, current alternatives, and technologies used in order to build the product.

Information design: Representing multidimensional data structures in a two-dimensional way is a non-trivial task. This process requires reasoning both analitically and visually. Because of this, information design depends on cognitive processes and visual perception for both encryption and decryption of this information.

Information design can be used to describe design practices in which the main objective is to inform. A sub-type of this discipline are infographics, which are data visualizations in which graphics communicate an information that could not be communicated otherwise.

Alternatives to the work presented:

1. Madrid's City Council Air Quality website: Madrid City Council's "Servicio de Calidad del Aire" (Service on Air Quality) offers a website in which anybody can look for the measurements for the air quality measurement stations. This website shows a graph for each station showing historical data, but it's limited: It doesn't show any other variables, or estimated measurements for a specific zone.

2. eltiempo.es offers a similar website with measurements for several variables and an Air Quality Index (from now on, AQI). However, it lacks of historical data, custom coordinates and other variables.

3. Google shows a widget about air quality on queries related to that subject. However, it is a simple widget and it doesn't have historical data nor other kinds of variables.

4. BreezoMeter, which is the source for the data in the previous Google widget, shows a heatmap with some other factors as well as the AQI. However, it lacks some specific variables that are used in this project, and the heatmap visualization could be confusing.

Technologies used in the project:

- Javascript: Interpreted programming language, object-oriented, based on prototypes, imperative, weakly-typed and dynamic.
- ECMAScript: Specification of the Javascript language. It has dynamic types lightly inspired by Java or C. It supports some features about object oriented programming by using objects with prototypes and pseudo-classes.
- Babel: Code transpiler for turning modern versions of ECMAScript code into legacy Javascript code.
- Node (or nodejs): Open-source, cross-platform JavaScript run-time environment based on ECMAScript, asynchronous and based on V8 engine by Google.
- MongoDB: NoSQL database system, document-oriented and open source.
- NoSQL: Non-relational database that saves data in documents similar to JSON.
- JSON: Light text format similar to Javascript objects. Easier to parse than XML.
- ESLint: Static code analyzer for checking styleguides in a programming language (in this case, Javascript).

- React: Javascript open source framework to create single page user interfaces.
- Turf: Javascript framework for spacial analysis (geo-data).

This project implements the MEAN or MERN technological stack, composed by MongoDB, Express, Angular/React and Nodejs. This stack works very well together and provides a full implementation of all the layers required by the fullstack application.

There are several layers in this stack that suit perfectly the layers of this proyect. There is an explanation for each layer (MERN - MongoDB, Express, React and Node) in relationship with the project, detailed below. The layers in the architecture itself of the project are detailed in the next section.

Because the main different between MEAN and MERN stacks is React or Angular for the front or visual side of the application, a comparison was needed in order to decide which one to use. In the end React, the frontend framework for the visual interface, was chosen over Angular on several relevant differences such as maintenability and integration. Other alternatives were Vuejs or Elm, but React was more extended and better solution in this case.

The express layer, or the HTTP(S) server, covers a necessity in which the web interface needed a way to communicate with the production database without dealing with the CRUD operations directly by itself. The express server provides a REST API for the application in order to handle data requests. Express was not the best solution performance-wise, but the server layer is so simple that it was prioritized easy integration and maintenance over performance.

Other alternatives for this layer could have been Ruby on Rails (Ruby) or Django (Python). For consistency purposes, express was better as it uses Javascript like the rest of the stack.

Regarding the database infrastructure used, there was a choice between noSQL and SQL databases. Non-relational databases (noSQL) are based on documents, key-value pairs, graphs, etc., so they have dynamic schemas to adapt to different data structures (or non-structured data). Some of these implementations were not useful as the data used in the project didn't

follow the structure of a graph, tuples, etc., but the dynamism and absence of relations made this option better in this case.

Some highlights about the differentes were mainly that SQL databases performs better than noSQL on complex queries, but noSQL databases perform better with hierarchical (nested) data.

The decission of using mongoDB (noSQL, document oriented database) was finally powered by the following considerations:

1. The structure of the public data for this project was non-relational.
2. The queries used in this project were not too complex.
3. The integration with the existing technological stack (MERN) is better.
4. The queries for manipulating the database follow a syntax similar to Javascript's.

When talking about the decissions made in the data collection scripts, it is important to highlight that the original project this work is based on was done in Javascript (Nodejs), so it was natural to follow this line of work.

The requirements for this layer were basically the capacity to connect to public servers to retrieve the data, the capacity to deal and process the data, and the capacity to modularize the scripts to integrate them with a wrapper and with the database.

The explanation of the layers implemented in the project can be found with more detail in the next section.

7.3. System architecture and design

This section describes the architecture of the project in terms of its many sections, one for each layer previously described. It also presents the specifications and requirements extracted from the analysis phase.

When adapting the previously explained layers to the present work, the following architecture must be considered:

1. A dashboard or web interface, where someone can look up data related to the air quality variables in Madrid, as well as traffic and meteorological data.
2. A crawler, or a set of scripts that collects public data from different sources with the purpose of providing the dashboard with the data needed.
3. A database for storing said data, in order to both serve data beyond the limits of the original data sources (which are given under a time span), and avoid the rate limits in the APIs.
4. A web server to provide the web interface with a REST API for consulting the database.

The aspects that are outside the reach of this project are mainly any kind of statistical analysis, predictions, machine learning or time series forecast on the data presented, as well as the capacity to show data on real time (because of limitations in the original data sources).

There are also several requirements that affect this project, from environment requirements regarding the development environment (such as the operating system used), to infrastructure and data presented to the user (such as uptime, language of the UI, accesibility, etc.)

The specifications regarding use cases of the UI are developed under the usual workflow of the application, considering interactions on each module.

In the design phase, the project followed a double diamond model to define requirements and specify goals to achieve with the users and the team involved. The four phases of this model (discovery, definition, development and delivery) help with providing a workflow to follow that will define a roadmap for the project. The adaptation of this model to the present work ended up to be as follows:

1. Requirements elicitation: Requirements obtained in research phase with the users intersect with the goals of the project to focus on potential and doable goals for the application.

2. Analysis: Requirements meet resources (data and time available) to define the scope of the project.

3. Design and prototyping: Even when the development phase of the double diamond model was done directly in software, some base designs were made to orient the development phase and the creation of the interfaces.

This design and prototyping phase took into account the logo design and the UI design (with the different submodules). When studying the environment, some decisions were taken when defining which information should be shown in each component. There was an investigation, both with the data sources (to know which data could be used) and with potential users of the application (to know which data could be useful).

Apart from the design considerations of the different components of the application, there was an aspect regarding hosting and deployment environment which had been taken into account.

Some alternatives were considered, such as AWS, Heroku or now.sh. AWS was the perfect candidate but the scalability of the application could also reflect an increase on the price of the infrastructure, so in the end the best solution found was to deploy the application resources locally.

About the interfaces of the components:

1. Crawler module: Uses Traffic, Meteo and Particles modules.

    a) Traffic: Gets XML data from data source, processes it into a Javascript object, then gets specific info from each datapoint, then calculates the median value for the traffic saturation in each zone, and returns the list of zones with values (and hour of the measurement) in a callback.

    b) Particles: Gets CSV data with particles for the present day, then pre-process the data cleaning columns, and translates every particle ID to its name. Then,

calculates the AQI of the given zone, and returns particles values, AQI and time in a callback.

c) Meteo: Gets JSON with data, then parses the data with the keys and IDs for the values obtained, to get a list of 4 stations with meteo values and hour of measurement. Then it maps the 24 zones of the application into its nearest meteo stations, obtaining meteo data for each AQI station. Finally, it returns the data in a callback.

d) Crawler: Calls all submodules, processes (joins) the given data, and saves it into the database.

2. Express module: Implements two routes in a REST API for the application, each of one has a different query for the database.

3. User interface module: It is divided into components, and it uses Webpack, Babel and other tecnhologies to transpile a final build into a folder used for deployment of the application. Those modules are the Web (main frame), Header (logo), Map (map component), Panel (with cards containing values) and Graph (with the historical data).

7.4. Evaluation

In order to evaluate the performance of the product, it was necessary to test the final product by all means possible. The evaluation of the project was divided into two main parts: Functional evaluation and usability testing. The functional evaluation could be divided into three main approaches or techniques: Unit testing, integration testing, and functional testing.

Integration tests are the base for functional testing. However, they depend on the isolation of the components, which could not be applied for most of the modules in this application. Because of this, it was necessary to use integration testing to check the correct functionality of the application with the external data sources, libraries, and database.

Regarding the usability testing on the application user interface, it followed a method by Jackob Nielsen[47] called "guerrilla usability testing". This method allows the examiner to test

the application with only 5 users, detecting up to 85 % of the usability errors in the interface. This method is composed by a series of interviews made to the participants, regarding profiling questions, visual questions, and tasks. These interviews were useful to provide a new set of requirements, which could be useful in future iterations of the product.

7.5. Work planning

The approach used in planning and organizing the different parts of this project was an agile software development approach based on the Scrum model. This approach is based on iterations of small versions of a working product, viewed as cycles or *sprints*. These iterations allow to try and test features and correct them if neccesary on the design phase of the next iteration. It is designed to be simple and efficient, when comparing it to the waterfall model used before in the industry.

The final approach used in this project was an adaptation of the original Scrum, mainly because the roles of the project were assigned to a single person, so most of the processes and artifacts of this methodology did not apply. This fact has advantages and disadvantages. For example, time and effort estimations were better because there was no need for coordination between team members, but there was too much diversification of profiles for this methodology.

When presenting all the data about this project, it is important to talk about costs. The costs estimated for the project were divided into three sections: Personal costs (in terms of people or roles), equipment costs (in terms of hardware and infrastructure used), and indirect costs (marginal, estimated utilities cost).

The personal costs were estimated taking into account average salaries for the roles assumed in the execution of this project. For this section, it is estimated 7000€ for the developer role, and 246€ for the designer role, which translates into 7246€ for this part.

The equipment costs were estimated by calculating the cost of the equipment used during the realization of this project and dividing it by the span of the project in terms of the lifespan of the equipment. Most of the costs were estimated to be 10 % of the original cost of the

hardware. For the main equipment, a laptop, it was 158€. The server costs were estimated to be 59€, and the rented infrastructure costs were 11.65€ and 8.93€, making a total of 237.58€.

The indirect costs were estimated to be a 20 % of the total, which was 7483.58€ so far. The indirect costs then made a total of 1496.72€, which then led us to the total cost of the project being the estimation of 8980.3€.

7.6. Conclusions

First of all, as this proyect started as a personal pet project, it was very satisfying to scale it to a bigger version, packed with more features.

Also, all the objectives initially set for the roadmap of this proyect were met:

1. Implementing a series of scripts to get public data related to this work
2. Implementing the infrastructure with the database and the server to save said data and have it available for the user at any moment in time
3. Create a visual user interface using React to show said data
4. Using design and development work methodologies to define the requirements of this proyect, design the UI, develop prototypes and evaluate them with users
5. Design and implement the whole application integrating all its modules, with global code style rules, and correctly packed and published
6. Write the present document, which provides a report for the creation process of this project, and a documentation for the work itself

As every aspect of the project was planned with high modularity and independence in mind, every layer of the project is flexible to its implementation, following the adequate functionality. This allows future developers to add new modules, or adapt current ones by adding functionality or updating the data sources.

Referencias

[1] A. VILCHES, D. GIL PÉREZ, J. C. TOSCANO y O. MACÍAS. (2014). Urbanización y Sostenibilidad, [En línea]. Disponible en: http://www.oei.es/decada/accion.php?accion=15.

[2] S. Bailén, «Vivienda y ciudad compacta. Concepto y debates sobre ecourbanismo en España.», *Cuadernos de Vivienda y Urbanismo, 10*, pp. 68-85, 2017. [En línea]. Disponible en: https://dx.doi.org/10.11144/javeriana.CVU-14.vccc.

[3] E. en Acción, «La calidad del aire en la ciudad de Madrid en 2017», Ecologistas en Acción, inf. téc., 16 de enero de 2018. [En línea]. Disponible en: https://www.ecologistasenaccion.org/IMG/pdf/info-calidad-aire-madrid-2017.pdf.

[4] S. Rueda, «El urbanismo ecológico», *Transformaciones urbanas sostenibles*, pp. 234-267, 2011.

[5] A. de Madrid, «Plan A: Plan de Calidad del Aire y Cambio Climático de la Ciudad de Madrid», inf. téc., 22 de septiembre de 2017. [En línea]. Disponible en: https://diario.madrid.es/aire/.

[6] O. M. de la Salud, «Guías de calidad del aire de la OMS relativas al material particulado, el ozono, el dióxido de nitrógeno y el dióxido de azufre», inf. téc., 2006. [En línea]. Disponible en: https://www.mapama.gob.es/es/calidad-y-evaluacion-ambiental/temas/atmosfera-y-calidad-del-aire/calidad-del-aire/salud/.

[7] A. E. de Meteorología, *Nota legal AEMET*. [En línea]. Disponible en: http://www.aemet.es/es/nota_legal.

[8] A. de Madrid, *Aviso Legal Portal de datos abiertos del Ayto. de Madrid*. [En línea]. Disponible en: https://datos.madrid.es/portal/site/egob/menuitem.400a817358ce98c34e937436a8a409a0/?vgnextoid=dab92f5fd9c91410VgnVCM100000171f5a0aRCR
vgnextchannel=dab92f5fd9c91410VgnVCM100000171f5a0aRCRD&vgnextfmt=default.

[9] ——, «PROTOCOLO DE MEDIDAS A ADOPTAR DURANTE EPISODIOS DE ALTA CONTAMINACIÓN POR DIÓXIDO DE NITRÓGENO», inf. téc.

[10] I. Meirelles, *Design for Information*. Parramón Arts y Design, 2014.

[11] S. K. Card, *Readings in information visualization*, Nachdr. San Francisco [u.a.]: Morgan Kaufmann, 2003.

[12] (). Portal web de Calidad del Aire del Ayuntamiento de Madrid, [En línea]. Disponible en: http://www.mambiente.munimadrid.es/sica/scripts/index.php.

[13] (). Página web de El tiempo (eltiempo.es), sección de calidad del aire, [En línea]. Disponible en: https://www.eltiempo.es/calidad-aire/madrid.

[14] (). Google, [En línea]. Disponible en: https://www.google.com/.

[15] (). BreezoMeter Air Quality and Air Pollution Map in Real Time, [En línea]. Disponible en: https://breezometer.com/air-quality-map/.

[16] (). BreezoMeter Air Quality API, [En línea]. Disponible en: https://breezometer.com/air-quality-api/.

[17] (). Standard ECMA-292, [En línea]. Disponible en: https://www.ecma-international.org/publications/standards/Ecma-262.htm.

[18] (). Babel - The compiler for the next generation JavaScript, [En línea]. Disponible en: https://babeljs.io/.

[19] (). Node.js, [En línea]. Disponible en: https://nodejs.org/en/.

[20] (). npm, [En línea]. Disponible en: https://www.npmjs.com/.

[21] (). MongoDB, [En línea]. Disponible en: https://www.mongodb.com/.

[22] (). ESLint - Pluggable JavaScript linter, [En línea]. Disponible en: https://eslint.org/.

[23] (). React - A JavaScript library for building user interfaces, [En línea]. Disponible en: https://reactjs.org/.

[24] (). Turf.js - Advanced geospatial analysis, [En línea]. Disponible en: http://turfjs.org/.

[25] A. Morgan. (26 de enero de 2017). Introducing the MEAN and MERN stacks, [En línea]. Disponible en: https://www.mongodb.com/blog/post/the-modern-application-stack-part-1-introducing-the-mean-stack.

[26] P. Krill. (15 de diciembre de 2016). Forget Angular 3, Google jumps straight to Angular 4, [En línea]. Disponible en: https://www.infoworld.com/article/3150716/application-development/forget-angular-3-google-skips-straight-to-angular-4.html.

[27] C. Cordle. (29 de junio de 2017). Why Angular 2/4 Is Too Little, Too Late, [En línea]. Disponible en: https://medium.com/@chriscordle/why-angular-2-4-is-too-little-too-late-ea86d7fa0bae.

[28] D. Abramov. (14 de junio de 2016). React Design Principles - Stability, [En línea]. Disponible en: https://reactjs.org/docs/design-principles.html#stability.

[29] E. Elliott. (feb. de 2016). Angular 2 vs React: The Ultimate Dance Off, [En línea]. Disponible en: https://medium.com/javascript-scene/angular-2-vs-react-the-ultimate-dance-off-60e7dfbc379c.

[30] M. Rybczonek. (23 de mayo de 2018). Why Is Vue.js Growing so fast? 4 Reasons Behind the Framework's Success, [En línea]. Disponible en: https://www.netguru.co/blog/why-is-vue.js-growing-so-fast-4-reasons-behind-the-frameworks-success.

[31] J. Schae. (31 de marzo de 2018). A Real-World Comparison of Front-End Frameworks with Benchmarks (2018 update), [En línea]. Disponible en: https://medium.freecodecamp.org/a-real-world-comparison-of-front-end-frameworks-with-benchmarks-2018-update-e5760fb4a962.

[32] R. Olah. (14 de abril de 2017). Elm vs. React: Development and Performance, [En línea]. Disponible en: https://www.codementor.io/rudolfolah/elm-vs-react-development-performance-compare-603dyh83m.

[33] (). CRUD Operations, [En línea]. Disponible en: http://mongodb.github.io/node-mongodb-native/3.1/tutorials/crud/.

[34] D. Tzur. (21 de marzo de 2016). The Unbelievable History of the Express JavaScript Framework, [En línea]. Disponible en: https://thefullstack.xyz/history-express-javascript-framework/.

[35] R. Santamaria Maso. (sep. de 2018). NO, you most probably don't need Express in your Node.js REST API, [En línea]. Disponible en: https://medium.com/car2godevs/there-are-expressjs-alternatives-590d14c58c1c.

[36] (). Ruby on Rails - A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern, [En línea]. Disponible en: https://rubyonrails.org/.

[37] (). Django - The web framework for perfectionists with deadlines, [En línea]. Disponible en: https://www.djangoproject.com/.

[38] (). SQL to MongoDB Mapping Chart, [En línea]. Disponible en: https://docs.mongodb.com/manual/reference/sql-comparison/.

[39] (). python, [En línea]. Disponible en: https://www.python.org/.

[40] (). csv — CSV File Reading and Writing (Python 3.8), [En línea]. Disponible en: https://docs.python.org/3.8/library/csv.html.

[41] D. Council, *Design methods for developing services*.

[42] ——, «A study of the design process», inf. téc.

[43] J. y. K. V. O'Grady, *manual de investigación para diseñadores*. Blume, 2018.

[44] A. Heckmann. (). Mongoose, [En línea]. Disponible en: https://www.npmjs.com/package/mongoose.

[45] J. Hartikainen. (nov. de 2015). What are Unit Testing, Integration Testing and Functional Testing?, [En línea]. Disponible en: https://codeutopia.net/blog/2015/04/11/what-are-unit-testing-integration-testing-and-functional-testing/.

[46] M. Olsson. (20 de julio de 2017). Javascript tools for end-to-end testing web applications, [En línea]. Disponible en: http://mo.github.io/2017/07/20/javascript-e2e-integration-testing.html.

[47] J. Nielsen. (19 de marzo de 2000). Why You Only Need to Test with 5 Users, [En línea]. Disponible en: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/.

[48] Wikipedia, *Scrum (desarrollo de software) — Wikipedia, La enciclopedia libre*, [Internet; descargado 18-septiembre-2018], 2018. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Scrum_(desarrollo_de_software)&oldid=110674954.

[49] Graphext. (nov. de 2018). Salary ranges for designers in Spain, [En línea]. Disponible en: https://twitter.com/graphext/status/1017091884672536577.

[50] C. Busquets. (). Hoja salarial anónima de gente que trabaja en el sector del desarrollo web, [En línea]. Disponible en: https://docs.google.com/spreadsheets/d/14pfsWFpanG-RWmqBZnEYVQFw_rL09kAaxqvBRYjx5lE/edit#gid=0.